Apple III

Business BASIC

Reference Manual Volume 2

Preface

Welcome to the second part of the Apple III Business BASIC manual. This volume contains a summary of Business BASIC and appendices with more technical information. The index at the end of this volume contains references for both volumes of the Business BASIC manual.

If you are new to programming in BASIC, or have not used Business BASIC before, you should first read the first volume of the Business BASIC manual for information about programming in Business BASIC and details of this dialect of the BASIC language.

7

Apple III Business BASIC References

Syntax Notation

The syntax of a language is a body of rules that define the various language elements and how they may be combined. There are simple elements that are combined into compound elements, which in turn can be combined into expressions and statements.

An element is defined like this:

> (element to be defined) ::= (some combination of previously defined elements).

Any uppercase letters or punctuation marks appearing on the right side of the definition must be typed exactly as shown. Lowercase letters represent free information that you must fill in. For example, in the definition

> goto statement ::= GOTO linenum

the letters "GOTO" must be typed just as shown, followed by any legal line number.

Some definitions have two or more lines containing ::= in them. These lines are equivalent definitions for a given key word.

To combine elements, the following symbols are used. Note that you do not type them when you are entering a program! They are for purposes of describing syntax only!

|	separate alternative elements.
[	enclose optional elements.
{ }	enclose repeatable elements that must occur at least once.

\ \ enclose elements whose values are to be used.
~ indicates that adjacent elements must be separated by delimiters. Delimiters are defined later.

Other characters found in the syntax descriptions are required by BASIC.

Here's an intuitive example of how this system is used to describe the various parts of BASIC's syntax. In this example, we'll use houses, as they are familiar places to most of us:

house

::= roof{door}{window}[fireplace][all-electric kitchen]

A house has a roof, one or more doors, one or more windows, and may have a fireplace and an all-electric kitchen.

home

::= house|cottage|mansion

A home can be a house, cottage, or mansion.

price

::= \house\

The selling price is the value of the house.

suburban neighborhood

::= house{~house}

Suburban neighborhoods have space between the houses.

urban neighborhood

::= house{house}

Urban neighborhoods have no space between houses.

The remainder of this chapter is a description of the statements and functions of BASIC.

Statements and Functions

ABS

::= ABS(aexpr)

```
)PRINT ABS(345)
345
)PRINT ABS(24-363)
339
```

Returns the absolute value of the argument; in other words, the value of the argument if it is positive, 0 if the value is zero, and the negative of the argument value if it is negative.

Arithmetic Operators

::= abop|auop

The operands of arithmetic expressions can be reals, integers, or long integers. (Long integers may not be mixed in expressions with either reals or integers.) There are nine arithmetic operators:

Symbol	Meaning	Example	Numeric Value
+	Unary plus	+5	+5
−	Unary minus	−2	−2
^	Exponentiation	2^4	16
*	Multiplication	4*6	24
/	Division	5/2	2.5
MOD	Modulo	7 MOD 5	2
DIV	Integer Division	7 DIV 5	1
+	Addition	4+7	11
−	Subtraction	9−2	7

ARRAYS

An *array* is an ordered collection of single variables, all of the same type. The name of the whole collection, called the *array name*, can be any legal variable name. The last character of the name determines the type of all of the variables in the array.

The individual variables (or *elements*) within an array are numbered, starting with number 0. To refer to any element within an array, you specify the name of the array, followed by the number of the element enclosed in parentheses, called a subscript. For example:

```
)PRINT AR(3)
)PRINT Prices(147)
)D%(0,0)=85
```

An array may have any number of elements up to the limit of available memory. The number of dimensions is the number of subscripts needed to specify an individual element within the array.

ASC

```
::= ASC(sexpr)
```

```
)PRINT ASC("BEEP")
)d$="Air horn" : PRINT ASC(d$+"s")
```

ASC returns the decimal ASCII code corresponding to the first character of the given string expression. If the string expression value is a null string, then the value −1 is returned.

ATN

::= ATN(aexpr)

```
)PRINT ATN(.3456)
.33275
)
```

Returns the arctangent, in radians, of the given argument. The value returned represents an angle in the range −pi/2 to +pi/2 radians.

BUTTON — see page 219

CATALOG

::= CAT[ALOG]

```
)CATALOG
)CATALOG /Apple1
)CATALOG /Apple1/Applekind
)CATALOG .D2
```

CATALOG displays a listing of a root directory or subdirectory specified by the pathname following the reserved word CATALOG. If the specified pathname is a given volume name, then the names of all files in the given root directory are displayed on the screen and the names of any subdirectories of the root directory are also displayed.

If no pathname is given, the pathname contained in PREFIX$ is assumed. CATALOG may be abbreviated as CAT.

If OUTPUT# is set to anything other than 0, the directory listing will not be sent to the screen, but to the specified file.

CHAIN

::= CHAIN pathname [,linenum]

CHAIN automatically loads and runs a specified program, without clearing the values of the variables left over from the previous program, or closing any files the previous program had left open. This allows variable values used in one program to be used in

another. The pathname of the program to chain must follow the reserved word CHAIN:

```
)CHAIN Lightning
```

If the chained program assigns dimensions to an array that was defined in the previous program, a ?REDIM ERROR occurs.

CHR$

```
::= CHR$(aexpr)

)PRINT CHR$(66.8)
)R$="68" : PRINT CHR$(VAL(R$))
```

Returns the ASCII characters corresponding to the value of the arithmetic argument which must be in the range 0 to 255, or an ?ILLEGAL QUANTITY ERROR occurs. See the appendix ASCII CHARACTER CODES.

CLEAR

```
::= CLEAR
```

CLEAR sets all numeric variables to zero, all strings variables to null strings, clears all BASIC pointers and stacks, and closes all open disk files except a file being executed. CLEAR has no options.

CLOSE and CLOSE#

```
::= CLOSE[# filenum]

)1000 CLOSE
)185 CLOSE# 1
```

Before ending program execution, all open files should be closed with either a CLOSE# or CLOSE statement. Any files closed during program execution must be reopened before they can be accessed again. Each time a file is opened, even if it was used earlier in the same program, BASIC assumes that the file has not previously been opened during the current execution of the program.

CLOSE# closes the file whose file number is equal to the arithmetic expression that follows CLOSE#.

CLOSE closes all files that are open when the statement is executed. All open files are also closed by a LOAD, CLEAR, NEW, or RUN statement. The CHAIN statement does not close any files.

CONT

::= CONT

CONT resumes execution of a program that has been halted by STOP, END, or CONTROL-C at the statement immediately following the one at which execution was suspended.

CONT does not clear the program, or reset the variables in memory, and there are no options associated with it.

A program halted by an error may be continued. BASIC will attempt to continue execution starting with the statement in which the error occurred. An error made in immediate execution will not prevent a program from being continued.

A program that has had any of its statements altered, or any new statements added, may not be continued. If you try, the

?CAN'T CONTINUE ERROR

message will be displayed. The values of variables in a program can be changed using assignment statements in immediate execution while the program is stopped.

CONTROL –5, 6, 7, 8, 9

Control–5, –6, –7, –8, or –9 entered *from the numeric keypad* all perform special functions.

Control-5 switches off the video refresh, allowing programs to run slightly faster. Pressing Control-5 again switches the video refresh back on.

CONTROL-6 clears the typeahead input buffer. Any characters you have typed on the keyboard that your Apple has not yet displayed are discarded.

CONTROL-7 suspends screen output until a second CONTROL-7 is pressed to restart it. The screen output buffer is not cleared.

CONTROL-7 has the side effect of suspending execution if it is pressed while characters are being displayed on the screen (that is, when the output buffer is in use).

CONTROL-8 causes all control characters sent to the screen to be ignored. Instead, the two-character abbreviations of the control characters are displayed. To make control characters function normally again, press CONTROL-8 once more. CONTROL-8 is useful in debugging programs. If you have a different character set loaded, you may see characters other than the abbreviations. See your Apple III Standard Device Drivers manual for a list of control character abbreviations.

CONTROL-9 clears the screen output buffer. All characters that would normally be displayed on the screen are not displayed. After a second CONTROL-9 is pressed, character display is resumed.

CONTROL-RESET

Pressing the RESET button while holding the CONTROL key down reboots your Apple III, just as if you had switched the Apple off and back on. Anything stored in memory is lost during a CONTROL-RESET (including your program and BASIC).

CONV

::= CONV(expr)

```
)G&=234234 | H&=523523 | PRINT CONV(H&-G&)
289289
)
```

Evaluates the argument, and returns a real value. The value may be assigned to a regular integer. The conversion from real to integer is automatic in the latter case. If CONV is used with a string expression, the effect is the same as with the VAL function.

CONV$

::= CONV$(expr)

```
)D%=345 | A%=453 | PRINT "a"+CONV$(D%*A%)+"z"
a156285z
)
```

Evaluates the given expression, and returns a string value.

CONV&

::=CONV&(expr)

```
)PRINT CONV&(2178-7954)
-5776
)PRINT CONV&("4.214")
4
)
```

Evaluates the given argument, and returns a long integer value.

If the argument is a string, then the effect is the same as using VAL followed by CONV& (see the chapter STRINGS AND STRING FUNCTIONS for an explanation of the VAL function). The value

returned must be within the range 922337203685775807 to −9223372036854775808, or an ?OVERFLOW ERROR occurs.

CONV%

::= CONV%(expr)

```
)PRINT CONV%(423.94)
424
)A&=7656 | B&=364 | PRINT CONV%(A&/B&)
21
)
```

Evaluates the argument and returns an integer value, rounding off to the nearest whole number. The value returned must be within the range −32768 to 32767, or

?OVERFLOW ERROR

is displayed.

COS

::= COS(aexpr)

```
)PRINT COS(1.571)
−2.03673E-04
)
```

Returns the cosine of an angle given in radians.

CREATE

::= CREATE pathname, CATALOG|TEXT|DATA [,aexpr]

```
)CREATE "/Pies/Applepie", Text
)CREATE Attache, Text, 4212
```

CREATE is used to make root directories, subdirectories, text files, and data files. You must specify exactly what type of file you want to create and its name, by following the reserved word CREATE with the

new pathname and either TEXT, DATA, or CATALOG, separated by a comma. TEXT specifies that a text file be created; DATA specifies that a data file be created; and CATALOG specifies a root directory or subdirectory.

A file's *record* size may be specified by appending an arithmetic expression to the CREATE argument list. The record size is required only for random-access files, and must be in the range from 3 to 32767. If no record size argument is given, the file record size defaults to 512 bytes. When creating subdirectories, the arithmetic expression specifies the size of the directory. The directory size defaults to 512 bytes if unspecified (enough to hold twelve files).

An attempt to create an already-existing file generates a ?DUPLICATE FILE ERROR.

DATA

::= DATA [literal|string|real|integer|long integer]
[{, [literal|string|real|integer|long integer]}]

```
)1158 DATA "Panjandrum",1.41421,Deficit&
```

Creates a list of elements that can be used by a READ statement.

DEF FN

::= DEF FN functionname (real variable) = aexpr

```
)10 DEF FN Negate(X) = -X
)20 DEF FN Sworded.4(C) = INT(RND(3)*100)
)30 DEF FN M5BY7.MAT(DED) = DED*LOG(33)-ABS(F%)
```

DEF FN allows you to define functions to be used in your programs. The function;s argument must be a real and the reserved words DEF FN must be followed with a defining expression. Only functions of type real can be defined. Integer variables may be used in the expression, but they are automatically converted to reals by BASIC.

An ?UNDEF;D FUNCTION ERROR occurs if a function is used in a statement before being defined.

DEL

::= DEL linenum [TO|,| –[linenum2]]

DEL deletes lines from the program stored in memory. You can specify either a single line or a range of lines to be deleted.

```
)DEL 7
)DEL 85 TO 515
)DEL 73, 193
)DEL 996-1010
```

Each of the examples above will delete all existing lines of the program presently in memory within the specfied range (including the single line).

In deferred execution, attempting to delete the line currently being executed or any smaller numbered lines causes a

```
?RANGE ERROR
```

message.

DELETE

::= DELETE pathname

```
)DELETE /Tree/Banana
```

The DELETE statement is used to remove the subdirectory or file specified as its argument. A subdirectory may be removed only if all files in that subdirectory have been deleted.

Even if all files in a root directory have been removed, you cannot remove the root directory.

A number of errors can occur with improper arguments appended to a DELETE statement. They are summarized in the table below.

Error Message	Cause
?VOLUME NOT FOUND ERROR	Volume name given does not exist.
?PATH NOT FOUND ERROR	Subdirectory does not exist.
?FILE NOT FOUND ERROR	Non-existent local file name.
?FILE LOCKED ERROR	Subdirectory contains files, or specified file is locked.
?WRITE PROTECTED ERROR	Diskette is write-protected.
?FILES BUSY ERROR	One or more files are open.

Table 7-1. Possible DELETE Errors.

DIM

::= DIM array variable name (aexpr)

You can allocate space for an array in your program with a DIM statement (DIM stands for dimension). For example,

```
)DIM MIND%(7,2,3)
)DIM Lights%(78,9), Bulbs$(2,45), Lanterns&(9,0,8), LY(16)
```

If you refer to an array before defining it with a DIM statement, BASIC automatically creates an array having 11 elements per dimension, with subscripts numbered from 0 to 10. If the statement

```
)PRINT D&(18,LOOP5)
```

is executed before the array D& is defined, a ?RANGE ERROR occurs, because the subscript 18 is greater than the default maximum of 10 for each dimension.

If the value of a subscript refers to either a nonexistent dimension or a nonexistent element (one that is greater than the highest numbered element in a given dimension), a ?BAD SUBSCRIPT ERROR occurs.

ELSE

::= ELSE expr|lexpr|linenum

See IF..THEN.

END

::= END

END is identical to STOP except that no message is displayed.

ENGRSPEC

::= [+|−] engrpart [fracpart] exp

```
)PRINT USING "+3#.4#4E"; 1729
+ 1.7290E+03
)PRINT USING "+3Z.4Z3E"; 1729
+01.729E+3
```

The engineering specification (engrspec) is closely related to the scispec. It forces the exponent;s value to be a multiple of 3, and has a *maximum* of three digit positions to the left of the decimal point.

Either #'s or Z's can be used to indicate digit positions, and their choice is significant only to the left of the decimal point: # replaces leading zeros with spaces, and Z prints leading zeros.

EOF

BASIC assigns the file reference number of the file causing an EOF error to the reserved variable EOF. You can then check the reserved variable EOF to determine the affected file.

When you use the reserved variable EOF in an ON. . .GOTO or ON. . .GOSUB statement, you must enclose EOF in parentheses. For example,

```
)ON (EOF) GOTO 100,200,300
```

ERR

When BASIC encounters an error, it assigns the reserved variable ERR a code number corresponding to the type of the detected error. You can then refer to the reserved variable ERR to determine what

kind of error occurred. For a list of these codes and the corresponding error messages, see the appendix ERRORS.

EXFN.

::= EXFN. pathname [(lespr|@var[{,lexpr|,@var}])]
)PRINT EXFN.CalcX(2)*32/256

EXFN. executes an external assembly language function (loaded by an INVOKE statement) that returns a real value.

If you want to pass an integer argument, you must precede the expression being passed with a percent sign.

To pass *addresses* of variables, precede the variable name with an at sign (@).

EXFN%

::= EXFN%. pathname [(lexpr:@var[{,lexpr:,@var}])]

EXFN%. is identical to EXFN. except that the assembly language function involved returns an integer instead of a real value.

If you want to pass an integer argument, you must precede the expression being passed with a percent sign.

To pass addresses of *variables*, precede the variable name with an at sign (@).

EXP

::= EXP(aexpr)

)PRINT EXP(3)
20.0855
)

Raises e (to 6 places, e=2.718282) to the power indicated by the argument value.

FIXSPEC

::=[**] [$] [+|-] digitspec
::=[**] [+|−] [$] digitspec
::=[**] [$] digitspec [+|−]
::=$$[+|−] digitspec
::=$$ digitspec [+|−]
::=[++|− −] [$] digitspec

+ reserves a character position for the sign. Sign is printed in all cases.

− reserves a character position for the sign. Sign is printed if negative; otherwise a space is printed.

$ reserves a character position for a dollar sign.

** means print asterisks instead of spaces in unused character positions.

++ reserves the rightmost unused position(s) for the sign (and following dollar sign, if any).

−− same as ++ except the sign is replaced by a space if it is positive.

$$ reserves the leftmost unused position(s) for a dollar sign (and following numeric sign, if any).

You cannot use $$, ++, or −− if you use Z for the digitspec.

```
)PRINT USING "+###.###"; 3.14159
+ 3.142
)PRINT USING "+6Z.3Z"; 09999
+009999.000
)PRINT USING "+6&.3&"; 09999
+ 9,999.000
)PRINT USING "**+6#.3#"; 09999
+**9999.000
)PRINT USING "**+6#.3#−"; 09999
***$9999,000
```

```
)PRINT USING "+$6#.2#"; 09999
+ $9999.00
)PRINT USING "$$+6#.3#"; 09999
$+9999.00
)PRINT USING "++6#.3#"; 09999
+9999.00
)PRINT USING "$--6#.3#"; 09999
$9999.00+
)PRINT USING "$$6#.3#+"; 09999
$ 9999.00+
```

The fixed-point specification (fixspec) controls the output format of fixed-point numbers with a PRINT USING or PRINT# USING statement. Fixed-point numbers are any numbers displayed without exponents, including integers, long integers, and real numbers.

A "digitspec", composed of combinations of the characters #, &, and Z, is used to define the format of the number being displayed.

A "#" reserves one numeric digit position. Leading zeros (if present) are replaced with spaces.

A "Z" reserves one numeric digit position, just like a "#", except that leading *zeros* are printed.

An "&" character reserves one position for a numeric digit or comma. Commas are inserted after every third digit left of the decimal point. Commas are included in the character count and leading zeros are replaced with spaces. At least five digit positions must be reserved to the left of the decimal point when using &.

If you specify a fixspec with a fractional part and apply it to an integer expression, only zeros will appear to the right of the decimal point, unless the SCALE function is used.

The entire field is filled with exclamation points if the number of digits displayed exceeds the number of digits specified to the left of the decimal point.

A "+" reserves a position for the sign of the number, and causes the sign to be printed in all cases, while a "−" causes the sign to be

printed only if the number is negative; a space is printed if it is positive. The sign of the number can also be placed after the last digit.

A "$" reserves a character position for a dollar sign. A pair of asterisks (**) causes asterisks to be printed instead of leading spaces when there are unused digit positions in the output field.

Note that the **, if used, must be the first thing in the fixspec and cannot be used if Z is used for the digitspec, because Z leaves no unused digit positions. The dollar sign may come next, or the number sign (+ or −).

FOR...NEXT

::= FOR control variable = aexpr1 TO aexpr2 [STEP aexpr3]
::= NEXT [control variable {,control variable}]

)15 FOR Index=1 to 500 : PRINT Card : NEXT Index

FOR and NEXT allow a group of statements to be executed a specified number of times. The first control variable given in the NEXT statement must be the same as the one named in the most recently executed FOR statement; the second control variable given must match the second most recently executed FOR statement, and so on. Incorrectly matched FOR and NEXT statements cause a ?NEXT WITHOUT FOR ERROR.

FRE

::= FRE

FRE is a reserved variable which stores the amount of remaining available memory, measured in bytes. FRE allows you to check on the amount of space remaining in memory. See the appendices VARIABLE MAPS and SPACE SAVERS, for information on using memory space more efficiently.

Each time you access FRE, string variable storage space is reorganized to recover unused space.

GET

::= GET var

)110 GET Press$

GET is used to assign a single character or numeral from the keyboard to a specified variable in your program, without displaying it on the screen and without requiring that the RETURN key be pressed.

CONTROL-C is treated by GET like any other character; it does not interrupt program execution.

If your program that uses GET was called up by an EXEC file, the input for the GET statement will be taken from the EXEC file instead of from the keyboard.

GOSUB

::= GOSUB linenum

)287 GOSUB 1158

Causes the program execution to branch to the indicated line. When a RETURN statement is encountered, execution branches to the first line following the most recently executed GOSUB statement.

Nesting subroutines more than 23 deep causes a ?STACK OVERFLOW ERROR.

GOTO

::= GOTO linenum

)GOTO 100
)100 GOTO 1158

GOTO causes program execution to branch to the indicated line number. You can also use it in immediate mode to begin executing a program presently in memory at a given point.

HEX

::= HEX$(aexpr)

```
)PRINT HEX$(780)
)PRINT HEX$(-1024)
```

HEX$ returns a four-character string that is the hexadecimal (base 16) equivalent of the value of the given arithmetic expression. The expression must be in the range −65535 to +65535, otherwise an ?ILLEGAL QUANTITY ERROR results.

HOME

::= HOME

HOME clears all text within the current text window and moves the cursor to the upper left corner of the window.

HPOS

See VPOS.

IF...GOTO and IF...THEN

::= IF lexpr GOTO linenum:statementlist
[:ELSE linenum:statementlist]
::= IF lexpr THEN linenum:statementlist [:ELSE linenum:statementlist]

```
)IF A=4 GOTO 473
)IF KP+BH GOTO 3785
)100 IF G& MOD F& >2 GOTO 121
)IF 0 THEN PRINT 1
)50 IF 2+2 THEN 2500
)IF S/3>=17 * NOT 2 THEN GOSUB 3000 : INVERSE : PRINT
"Hi"
)IF X=1 THEN Y=2 : ELSE Y=3
)IF 3<PL5 THEN PL5=-PL5 : ELSE NORMAL : GOTO 376
)718 TEXT : IF NOT Y THEN 3200 : ELSE WINDOW 1,1 TO 4,4
    : GOTO 457
```

If the expression following IF evaluates to non-zero, the instructions following THEN or GOTO in the same line will be executed. If the expression evaluates to false (zero) then execution will continue with the next higher-numbered line.

In an IF...THEN statement, the instructions can be any line number to which execution should branch, or a statement list for BASIC to execute.

In an IF...GOTO statement, the instruction must be a line number to which execution can branch.

The optional ELSE clause in IF...THEN statements allows you to specify instructions for BASIC to execute if the truth value of the logical expression is false. In other words, when the expression is false, instead of having execution pass to the next higher numbered line, you can have BASIC execute some instructions. The instructions following the reserved word ELSE can be a line number to which execution should branch, or a statement list to execute. If the logical expression is true, the ELSE clause and any statements following on that line are ignored.

INDENT

INDENT is a reserved variable that contains the number of spaces to be used to indent FOR..NEXT loops in the program listings. Its default value is 2.

INPUT

```
::= INPUT [string,|;] var{,var}
)1000 INPUT INPUT Zoo$,Gnus,Tolls
)20 INPUT "Enter your age in years"; AGE
```

INPUT accepts numbers or text typed at the keyboard and assigns their values to variables specified in the INPUT statement. INPUT can be used in deferred execution only.

You may optionally include a string in an INPUT statement. The optional string must be a sequence of characters in quotation marks, followed by a comma or semicolon; it cannot be a string variable or expression. When the optional string is present, it is displayed exactly as specified; no question mark, spaces, or other punctuation are displayed after the string. You can use only one optional string.

You can halt program execution during an INPUT statement by typing CONTROL-C as the first character of your response, followed by return.

If only the RETURN key is typed when a string response is expected, the response is interpreted as a null string.

INPUT#

```
0 ::= INPUT# filenum # [,recnum] # [;var] # {,var}]]

)INPUT# 2; Payment%, Grease$
)INPUT# 8, 34; DG(0), DG(2), DG(4)
```

INPUT# reads a line of text for each variable in its list of variables. The file to be read from is defined by a file number following the reserved word INPUT#. If the file number is followed by a comma, the arithmetic expression following the comma specifies a record number at which to begin file access.

INSTR

```
::= INSTR(sexpr, sexpr # [,aexpr])

)PRINT INSTR("Rain in Spain on the Plain", "ai")
2
)PRINT INSTR("Rain in Spain on the Plain"; "ai", 5)
11
```

INSTR searches for occurrences of a specified substring within a string and returns the number of the first character of the substring.

The optional arithmetic expression specifies the character position where the search should begin. If no arithmetic expression is

specified, the search begins with the first character of the string expression. If the search fails, 0 is returned.

If the arithmetic expression is greater than the length of the string expression or less than 1, then an ?ILLEGAL QUANTITY ERROR occurs.

INT

::= INT(aexpr)

```
)PRINT INT(3.3)
3
)X=INT(-3.3) : PRINT X
-4
)
```

Returns the largest whole number value less than or equal to the argument value.

Notice that we said "whole number" in the last sentence, and deliberately avoided the term "integer". This is because the INT function actually returns a real number

INTEGER

::= #[+|-]{digit}

An *integer* is any positive or negative whole number without a decimal point, from −32768 to 32767. Attempting to assign a value beyond this range to an integer variable generates the ?ILLEGAL QUANTITY ERROR message.

INVERSE

::= INVERSE

INVERSE sets all subsequent display to black letters on a white background. Characters on the screen before the execution of the INVERSE or NORMAL statement are not affected.

INVERSE has no effect on characters read from or written to files.

INVOKE

::= INVOKE pathname #[{,pathname}]

```
)INVOKE FP1, FP2, /Floppy2/Subr/FP3
)INVOKE Fastprint
```

INVOKE loads a file containing specified assembly language subroutines into memory.

You may load as many subroutines at once as you like by separating each file's pathname by commas.

Executing INVOKE effectively erases any external subroutines previously loaded by other INVOKE statements and returns any unused memory to BASIC.

KBD

```
ON (KBD)-64 GOTO 100,200,300
```

KBD contains the ASCII value of the last key struck (see the appendix, ASCII CHARACTER CODES).

When you use the reserved variable KBD in an ON...GOTO or ON...GOSUB statement, you must enclose KBD in parentheses, or BASIC will not treat it as a variable.

LEFT$

::= LEFT$(sexpr, aexpr)

```
)PRINT LEFT$("Appleskin",5)
Apple
)PRINT LEFT$("Sparkling",3)
Spa
```

LEFT$ returns a string of specified length composed of the leftmost characters of the given string expression.

If the value of the arithmetic expression exceeds the length of the string expression value, all of the characters of the string expression value are returned. If the string expression value contains more than 255 characters, a ?STRING TOO LONG ERROR results. The value of the arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the range 1 to 255, or an ?ILLEGAL QUANTITY ERROR results.

LEN

::= LEN(sexpr)

```
)PRINT LEN("ABCD")
)PRINT LEN(Yarn$)
```

LEN returns an integer value equal to the length of the string expression, in the range 0 to 255. A string expression containing more than 255 characters causes a ?STRING TOO LONG ERROR.

LET

::= #[LET] var|modifiable resvar = \expression \

```
)LET Henry=FatherofJack
)LET WaterAnimal$="Blue whale"
```

The variable name to the left of the "=" is assigned the value of the expression to the right of the "=". Only one assignment may occur per statement. LET is optional.

LIST

::= LIST # [linenum] # [TO|,|-# [linenum2]]

```
)LIST
)LIST 5 TO 300
)LIST 5,300
)LIST 5-300
)LIST TO 2100
)LIST 1585 TO
```

LIST displays the contents of the program currently in memory.

The first example above displays the entire program presently in memory. The next three display line 5 through 300, inclusive (assuming that they exist) of the program presently in memory.

The last two examples will list, respectively, from the beginning of the program to line 2100 and from line 1585 to the end. The word "TO" in all examples may be replaced with either a "–" or a ",".

The listing can be stopped and restarted by repeatedly pressing CONTROL-7 (on the numeric keypad). Pressing CONTROL-C terminates the listing.

LOAD

::= LOAD pathname

)LOAD Countdown
)LOAD .D2/Somefile

LOAD reads a specified BASIC program from a disk file, and stores it in memory. The pathname of the program to be loaded must follow the reserved word LOAD (see the chapter FILE I/O for an explanation of pathnames).

All variables in the loaded program are cleared; numeric variables are all set to zero, string variables are set to null strings. All files are closed, with the exception of any EXEC file being executed. Any existing program is cleared from memory.

Attempting to load a file other than a BASIC program causes a ?TYPE MISMATCH ERROR.

LOCK and UNLOCK

::= LOCK pathname
::= UNLOCK pathname

)LOCK Barndoor
)UNLOCK Secrets

LOCK prohibits writing to, saving, or deleting the file named as its argument. Locked files are shown with an asterisk to the left of their file type when cataloged. Volume names may not be locked, but subdirectories may be.

UNLOCK allows you to "unprotect" a locked file that you want to delete, rename, change, or save. The reserved word UNLOCK must be followed by the file's name.

LOG

::= LOG(aexpr)

```
)PRINT LOG(20.0855)
3
)
```

Returns the natural (base e) logarithm of the argument value.

LOGICAL EXPRESSIONS

Logical expressions are also called relational expressions and Boolean expressions. They are similar to arithmetic expressions, but use different operators. A logical expression can have only one of two values: 1 (or true) or 0 (or false). Any arithmetic expression with a non-zero value has a truth value of 1, and any with a value equal to zero has a truth value of 0.

There are nine logical operators:

Symbol	Meaning	Example	Truth value
=	Equal to	3=3	1
<	Less than	3<1	0
>	Greater than	7>4	1
<= or =<	Less than or equal to	5<=4	0
>= or =>	Greater than or equal to	8>=5	1
<> or ><	Not equal to	4<>4	0
AND	Conjunction	5 AND 0	0
OR	Inclusive disjunction	8 OR 3	1
NOT	Negation	NOT 4	0

It is possible to use logical operators in string expressions. For example, "alpha" < "beta" is true.

LONG INTEGER

::= [+|−]{digit}

Long integers may be up to 19 digits long. They may not be mixed in arithmetic expressions with regular integers or reals (described below). Long integer variable names must end with an ampersand (&). A long integer can range from −9223372036854775808 to 9223372036854775807. Exceeding this range causes the message

?OVERFLOW ERROR

to be displayed. Entering the number −9223372036854775808 from the keyboard will also cause the same error.

MID$

::= MID$ (sexpr, aexpr1 [, aexpr2])

```
)PRINT MID$("Bookkeeping",5)
)keeping
)PRINT MID$("Bookkeeping",5,4)
)keep
```

MID$ returns a substring of a given string expression. The first arithmetic expression specifies the first character to be returned from the string, and the second arithmetic expression (if given) specifies the length of the substring to be returned.

If the value of the first arithmetic expression exceeds the length of the string expression value, then a null string is returned. If the value of the second arithmetic expression specifies a greater number of characters to be retrieved from the string expression value than exist, all of the characters from the position specified by the value of the first arithmetic expression to the end of the value of the string expression are returned.

If the string expression value contains more than 255 characters, a ?STRING TOO LONG ERROR occurs. If the value of either arithmetic expression is outside the range 1 to 255, then an ?ILLEGAL QUANTITY ERROR occurs.

NEW

::= NEW

NEW erases the current program and all its associated variables from the Apple's memory, and closes all open files except a text file being executed (see the EXEC statement in the chapter AUTOMATIC EXECUTION). NEW has no options.

NORMAL

::= NORMAL

This is the default display mode. NORMAL sets the display to white letters on a black background. Characters on the screen before the execution of the NORMAL statement are not affected.

NORMAL has no effect on characters read from or written to files.

NOTRACE

::= NOTRACE

There are no options associated with NOTRACE, it simply cancels TRACE; the line numbers of executing program statements are not displayed.

OFF EOF#

::= OFF EOF# filenum

The OFF EOF# statement cancels an ON EOF# statement. After an OFF EOF# statement has been executed, BASIC resumes displaying error messages and halting execution when an end of file is reached, just as it did before the ON EOF# statement was executed. You must

follow the reserved word EOF# with a file reference number to specify which file's ON EOF# statement should be canceled.

ON EOF#

::= ON EOF# filenum statementlist

ON EOF# is used to force BASIC to allow your program to control what happens if BASIC reads past the end of a file, just as an ON ERR statement allows your program to perform its own error handling. EOF stands for End Of File.

A statement or statement list must follow the reserved word EOF#, exactly like a GOTO statement.

ON ERR and OFF ERR

::= ON ERR statementlist
::= OFF ERR

```
10 REM EXAMPLE OF ERROR HANDLING
20 ON ERR GOSUB 1000
30 INPUT "Please type a single number between 1 and 100
";X
40 PRINT "The number you typed was ";X
50 END
1000 REM ERROR HANDLING SUBROUTINE
1010 PRINT : PRINT "I'm very sorry, but only a number
will do. Please try again."
1020 RETURN
```

ON ERR is used to force BASIC to let your program handle any errors that might occur by branching to an error-handling subroutine that you have included in the program.

The RETURN at the end of the error-handling routine causes execution to branch back to the line where the error occurred.

The ON ERR statement should not be used as a tool for finding errors in programs. (Use the TRACE statement for this instead.)

If a program contains more than one ON ERR statement, only the most recently executed one will be used.

OFF ERR cancels the most recently executed ON ERR statement. There are no parameters or options associated with this statement. After an OFF ERR statement has been executed, BASIC resumes displaying error messages and halting execution just as it did before the ON ERR statement was executed.

The statements that ON ERR causes to be executed must themselves be free of errors, or an endless loop may result. The endless loop will be unstoppable by CONTROL-C, because CONTROL-C is itself considered an error. For a complete list of BASIC errors, see the appendix ERRORS.

ON KBD and OFF KBD

::= ON KBD statementlist
::= OFF KBD

```
10 ON KBD GOTO 100 : REM BASIC branches here when any key is pressed
20 PRINT "."; : REM Print periods while not handling key-strokes
30 GOTO 20
100 PRINT KBD : REM Display the ASCII value of the key last pressed
110 ON KBD GOTO 100 : REM Reenable ON KBD. Must be before return
120 RETURN : REM Program jumps back to the statement following the one during which a key was pressed
```

ON KBD is used to cause BASIC to execute a specific statement list immediately when any key is pressed. The statement list to be executed must follow the reserved word KBD.

Note that you must reenable the ON KBD statement immediately before executing the RETURN statement at the end of the execution list.

After an ON KBD statement has been executed, BASIC continues executing the program normally—but as soon as any key is pressed, execution branches back to the most recently executed ON KBD statement. Then the statement list pointed to by the ON KBD statement is executed.

The branch to the ON KBD statement list is treated as a GOSUB to a subroutine, so the program segment that KBD causes to be executed must end with a RETURN statement. To enable ON KBD to handle more than one keystroke, the last statement in the list should be another ON KBD statement.

When ON KBD is in effect, the program can not be halted with CONTROL-C, since this is treated just like any other keystroke. However, the ON KBD statement could cause a branch to a STOP or END statement if a CONTROL-C is typed. A RETURN statement placed after the STOP would allow the CONT statement to be used.

ON...GOSUB

::= ON aexpr GOSUB linenum {[,linenum]}

)1000 ON Corfu GOSUB 1000, 2000, 3000, 4000

ON...GOSUB is identical to the ON...GOTO statement, except that the line numbers following the reserved word GOSUB must be line numbers of subroutine entry points.

ON...GOTO

::= ON aexpr GOTO linenum {[,linenum]}

)1000 ON X GOTO 100, 10, 300, 40

ON...GOTO is used to specify different program branch points, based on the value of an arithmetic expression. The arithmetic expression

must follow the reserved word ON, and the line numbers to which execution branches must follow the reserved word GOTO.

If X=1, execution branches to the first line in the list of numbers (line 100). If the value of X is 2, then execution branches to the second line in the list (line 10). If X=3, execution branches to line 300 (the third line in the list), and so on.

The value of the arithmetic expression must be within the range 0 to 255, or an ?ILLEGAL QUANTITY ERROR occurs. If the value of the arithmetic expression is 0, or greater than the number of line numbers given in the ON . . . GOTO statements, the list of line numbers is ignored, and execution continues with the next statement in the program.

OPEN#

::= OPEN# filenum [AS INPUT|AS OUTPUT|AS EXTENSION], pathname [, recsize]

```
)OPEN #6, Door
)OPEN #4, Window, 163
)OPEN #2 AS INPUT, .CONSOLE
)OPEN #1 AS OUTPUT, .PRINTER
)OPEN #1 AS EXTENSION, Ladder
```

OPEN# is used to open files for access, and must precede any file I/O statements accessing a given file. The arguments following OPEN# are a file reference number and the file's pathname. The file reference number is used in all subsequent I/O statements to refer to the file while it is open. The file reference number can be any arithmetic expression having a value between 1 and 10, inclusive.

If the OPEN# file reference number is followed by the reserved words AS INPUT, the file is opened as a read-only file and may not be written to.

If the OPEN# file reference number is followed by the reserved words AS OUTPUT, the file is opened as a write-only file and may not be read from.

The AS EXTENSION option is a variant of AS OUTPUT, and is used in sequential access to allow PRINT# or WRITE# statements to append new information to the end of an existing file without disturbing any data that was put in the file earlier.

OUTPUT#

::= OUTPUT# filenum

)OUTPUT #5

OUTPUT# redirects screen output to a specified file. All PRINT, LIST, TRACE, and CATALOG statement output is sent to the specified file, but keyboard input is echoed and error messages are still sent to the screen. The file used for output is specified by its file number (set by an OPEN# statement) following the reserved word OUTPUT#.

Remember that system I/O devices such as .CONSOLE and .PRINTER are treated as files and may be opened and used as such.

To resume normal screen output, type

)OUTPUT# 0

and characters will again be displayed on the screen. A CLOSE or CLOSE# statement also redirects output to the screen.

OUTREC

OUTREC is a reserved variable that contains the maximum length of lines output by the LIST command. The value of OUTREC must be greater than the value of INDENT.

PDL — see page 219

PERFORM

::= PERFORM pathname [(lexpr|@var[{,lexpr|,@var}])]

)PERFORM StrangeRites(&Pennies, %Accountants)
)PERFORM Errproc(R,13-6,@D)

PERFORM executes a specified assembly language procedure previously loaded by an INVOKE statement. If an argument list is

present (enclosed in parentheses after the procedure name) each argument is evaluated and passed to the procedure before execution.

To pass real numbers or the values of single variables, just include them in the argument list as an expression (for an explanation of expressions see the chapter EXPRESSIONS AND STATEMENTS).

If you want to pass an integer argument, you must precede the expression being passed with a % sign.

To pass *addresses* of variables, precede the variable name with an @ sign.

If you want your subroutine to operate on a string in memory, using the at sign gives an address pointing to the string's descriptor in memory. The subroutine should be designed to act on the string from that point on.

POP

::= POP

POP allows you to jump out of one level of subroutine nesting by removing the top pointer from the program stack and discarding it. When the next RETURN statement is encountered after a POP statement is executed, instead of branching to the first statement beyond the most recently executed GOSUB, BASIC branches to the first statement beyond the second most recently executed GOSUB.

PREFIX$

PREFIX$ is a reserved variable that contains the most recently assigned default pathname prefix.

PRINT

::= ?|PRINT {[,|;] [expr]} [,|;]

)PRINT
)PRINT "Several words of text."
Several words of text.

```
)A$="E is about " : E=2.718
)PRINT A$;E
E is about 2.718
```

PRINT displays text. An item list may include any expression, comma, semicolon, TAB specification, or SPC specification following the reserved word PRINT.

Expressions in PRINT statements are evaluated and their values displayed. If there are several expressions, their values are displayed in sequence. A PRINT statement without an item list moves the cursor to the beginning of the next screen line.

If a comma separates two expressions, a tab action separates their values on the screen; if a semicolon separates them, the second value is displayed after the first with no intervening spaces.

Following the last expression in a PRINT statement, there may be a semicolon, comma, or nothing. If there is nothing after the last expression, the cursor moves to the beginning of the next screen line. A comma causes a tab action. A semicolon leaves the cursor in the position immediately following the last character displayed.

PRINT USING

```
::= ?|PRINT filenum [, recnum] USING linenum|string|svar
[; expr[{,expr}]] [;]
```

The PRINT USING statement is the same as a PRINT statement with a USING clause used to control the format of information sent to the file. Both PRINT and USING are described in detail in the chapter on APPLE BUSINESS BASIC I/O.

PRINT#

```
[::= ?#|PRINT# filenum [, recnum] [; expr [ . .; expr] . [;] ]

)PRINT# 1; W$(0,0,0), LEFT$(W$(0,0,1))
)PRINT# 10, 4755; A&+24, T&/43, R%
```

PRINT# writes information to files like PRINT writes information to the screen. Its syntax is the same as the PRINT statement described above. After the file number (or record number, if included) a list of expressions separated by commas must follow.

One line of text is written for each expression in the list. PRINT# automatically performs any necessary numeric to string type conversions (similar to the STR$ function) in order to transfer the information from the expressions to the file.

A PRINT# statement in which a specific record number is given starts writing information to the file at the beginning of the specified record.

The SPC specification can be used with PRINT# statements in the same way it is used with PRINT statements.

PRINT# USING

::= ?#|PRINT# filenum [, recnum] USING linenum|string|svar [; expr[{,expr}]] [;]

The PRINT# USING statement is the same as a PRINT statement with a USING clause used to control the format of information sent to the file. Both PRINT and USING are described in detail in the chapter on APPLE BUSINESS BASIC I/O.

READ

::= READ var [{,var}]
)2001 READ Odyssey$,Wine%,Dark&,C

READ assigns the variables in its list values taken from elements in the program's DATA statement list.

If CONTROL-C is a data element, it does not halt execution of the program even when it is the first character of an element. With this exception, data elements read into string variables follow the rules for INPUT responses assigned to string variables:

- Either literal or quoted strings may be used.
- Spaces preceding and following strings are always ignored, except with quoted strings.

- Quotation marks appearing within a quoted string cause the

  ?SYNTAX ERROR

  message, but all other characters, including commas are accepted as characters in that string. The entire string may be enclosed within quotes, however.

- The colon and comma are accepted only in quote-enclosed strings.

If a READ statement attempts to assign a string data element value to an arithmetic variable, the

?SYNTAX ERROR

message appears when the incorrect value type is assigned.

Variables are assigned values of zero or null string (depending on the variable's type) when any of the following conditions are met when an attempt is made to read a data element:

- A comma is the first non-space character following the reserved word DATA.

- There is no data element between two commas.

- The last character in a DATA statement is a comma (when the comma is being read as a data element).

READ#

::= READ# filenum [, recnum] [; var[{,var}]]

)READ# 7; Pip1, Pip2
)READ# 8, 54; Twelve%, Strong&(2)

READ# gets information from a data file specified by its file number. An optional record number may be included to specify a particular record in a random-access file to begin reading. A variable list following the file number (and optional record number, if included) defines where to put the information being read.

The following table defines the conversion limits of the READ# statement:

Variable	Data Field Type	Result
Real	Real	OK
"	Integer	OK
"	Long Integer	OK; Possible loss of accuracy
"	String	?TYPE MISMATCH ERROR
Integer	Real	OK in the range +-32767
"	Integer	OK
"	Long Integer	OK in the range +-32767
"	String	?TYPE MISMATCH ERROR
Long Integer	Real	?OVERFLOW ERROR if > +-E18
"	Integer	OK
"	Long Integer	OK
"	String	?TYPE MISMATCH ERROR
String	Real	?TYPE MISMATCH ERROR
"	Integer	?TYPE MISMATCH ERROR
"	Long Integer	?TYPE MISMATCH ERROR
"	String	OK

Table 7-2. READ# Statement Conversion Limits.

REALS

::= [+|−]{digit}[.{digit}][E[+|−]digit[digit]]

::= [+|-][digit].[{digit}][E[=|-]digit[digit]]

A *real* is any positive or negative number, and may have a fractional part. A numeric constant with a decimal point is always of type real, even if it has only zeros to the right of the decimal point. However, not all reals must have decimal points.

Reals whose absolute values are greater than or equal to .01 and less than 999999.2 are expressed in conventional notation. For example, 1, +1, −1., 3.14, 999.999, and −0.2 are all real numbers expressed in conventional notation.

A real may also be expressed in scientific notation, such as 3.3E2, −3.3E4, 3.3E-4, or −3.3E-3. The real number 5.3E12, for example, is equal to 5.3 times 10 raised to the 12th power.

Here are examples of conventional notation vs. scientific notation.

Conventional Notation	**Scientific (E) Notation**
300	3E2 = 3*(10^2)
320	3.2E2 = 3.2*(10^2)
.44	4.4E-1 = 4.4*(10^−1)
−.033	−3.3E-2 = 3.3*(10^−2)
1000000000000	1E12 = 1*(10^12)

Reals must be within the range -1.7E38 to 1.7E38 or an ?OVERFLOW ERROR occurs.

REC

::= REC(filenum)

REC returns the current record number of the file specified by the value of the arithmetic expression following the reserved word REC.

If you use the INPUT# or READ# statements to access the catalog of a directory, REC returns the number of the line currently being accessed.

REC has the same error conditions as the TYP function.

REM

::= REM anything

)100 REM This can be a lifesaver.

The reserved word REM must be the first thing in a remark statement or the statement will not be treated as a remark. REM statements must not exceed 250 characters in length. If you comment your programs heavily, use several REM statements in successive lines rather than using one very long remark.

RENAME

::= RENAME pathname1, pathname2

)RENAME /Floppy2/Animals/Dogs, /Floppy2/Animals/Pigs

RENAME is used to change the names of volumes, subdirectories, and local files. RENAME's argument list is composed of the old pathname, followed by a comma, followed by the new pathname.

You cannot use the RENAME statement to create a file or subdirectory, only to rename an existing one. Use the CREATE statement to make new files and root directories.

A local filename or subdirectory may not be changed to another volume name or subdirectory.

RESET

Pressing the RESET button is equivalent to pressing CONTROL-C, except that RESET clears some program stacks and pointers, and you cannot use an ON KBD or ON ERROR statement to handle RESET.

RESTORE

::= RESTORE

RESTORE moves the data list pointer back to the beginning of the data list, allowing you to read the same data more than once. RESTORE has no parameters or options.

RETURN

```
::= RETURN
```

RETURN has no parameters or options. When executing a RETURN statement, BASIC removes one pointer from the top of the *program stack* and branches to the statement indicated by the pointer. This is the statement immediately following the most recently executed GOSUB statement, unless a POP statement has been executed since the most recent GOSUB was encountered.

If BASIC attempts to execute one more RETURN statement than it has pointers on the program stack, the ?RETURN WITHOUT GOSUB ERROR occurs.

RIGHT$

```
::= RIGHT$(sexpr, aexpr)

)PRINT RIGHT$("Appleskin" + "Ware", 8)
skinWare
)B$=RIGHT$("Fruitbat", 3) : PRINT B$
bat
```

RIGHT$ returns a string of specified length composed of the rightmost characters of the given string expression.

RND

```
::= RND(aexpr)

)PRINT RND(8)
.830965
)
```

The RND function returns a random real positive number less than 1.

RND generates a new random number each time it is used if the argument value is greater than zero.

RUN

::= RUN [pathname[, linenum]]|[linenum]

```
)RUN
)RUN 205
)RUN Marathon
)RUN Assets, 7254
```

RUN is used to start running a program. When a RUN statement is entered, BASIC clears all variables, closes all open files except executing text files, and begins to execute the program in memory beginning with its smallest line number, or at the line number indicated. A program on disk can be run by following RUN with the program's pathname.

If you specify a non-existent line number, an ?UNDEF'D STATEMENT ERROR appears. If the file you specify is not found after searching the disk, a ?FILE NOT FOUND ERROR occurs.

SAVE

::= SAVE pathname

SAVE writes a copy of the program currently in memory to a disk file. You must specify the file to be saved by following the reserved word SAVE with a pathname. If there is already a BASIC program with the same pathname on the disk, it will be overwritten and lost. If a locked BASIC program with the same name is on the disk, you will get a ?FILE LOCKED ERROR. If a file on the disk having the specified name is not a BASIC program, a ?TYPE MISMATCH ERROR occurs.

SCALE

::= SCALE(variable name, aexpr)

```
)A&=12345678901234567
)PRINT USING "$$20&#.##";SCALE(-2,A&)
$123,456,789,012,345.67
```

SCALE is used in conjunction with PRINT USING to shift the decimal point of a displayed value to the left or the right. SCALE uses two arithmetic expressions as arguments. The first argument defines the number of places to the right that the decimal point should be moved. The second argument is the actual numeric value to be output.

The resulting exponent of the value must be between −99 and +99, or an ?ILLEGAL QUANTITY ERROR occurs.

SCISPEC

::= [+|-] [scipart] [fracpart] exp

```
)PRINT USING "+#.4#4E"; 3.1415926
+3.1416E+00
)PRINT USING "+.4#4E"; 3.1415926
+.3142E+01
```

The scientific-notation specification (scispec) formats numeric output in scientific notation. The scispec is simpler than the fixspec, having either one digit or none to the left of the decimal point.

"#" characters, either stated explicitly or by a repeat factor define the number of digits to the right of the decimal point. The exponent position is defined with the letter E, and a repeat factor is legal.

Either three or four character positions *must* be allowed for the exponent.

When the spec calls for one digit position to the left of the decimal point, the first significant digit of the value is placed there; when there is no digit position to the left of the decimal point, the most

significant digit is placed to the right of the decimal point. In either case, the exponent is then calculated to make the displayed value correct.

SGN

```
::= SGN(aexpr)

)PRINT SGN(-234)
-1
)PRINT SGN(2496+234)
1
)PRINT SGN(5E4-5E4)
0
)
```

Returns −1 if the argument value is negative, returns 0 if the value of the argument equals 0, and returns 1 if the argument value is positive.

SIN

```
::= SIN(aexpr)

)PRINT SIN(2.718)
.411038
)
```

Returns the sine of an angle given in radians.

SPC

```
::= SPC(aexpr)

)PRINT "A"; SPC(1); "B"; SPC(2); "C"
A B  C
)PRINT "D"; SPC(5); "E"; SPC(5); "F"
D    E    F
)SPC(250)SPC(139)SPC(255)
```

SPC is used in PRINT statements to define (by the expression enclosed in parentheses) the number of spaces to be inserted after the last-printed character.

Each SPC statement is limited to a maximum value of 255, but you can place as many spaces as you wish by stringing together a series of SPC statements.

SQR

::= SQR(aexpr)

```
)PRINT SQR(32+42)
5
)
```

Returns the positive square root of the argument value.

STEP

::= STEP(aexpr)

```
)2000 FOR Farenheit=1 to 451 STEP 3 : NEXT
)2005 PRINT "Fire!!!"
)87 FOR Counter=10 to -10 STEP -1. . .NEXT Counter
```

STEP allows you to increment (or decrement) the control variable of a FOR...NEXT loop (described earlier) by integer steps other than 1. If a negative value is specified in a STEP clause, the loop counts backwards. If a positive value is specified, the loop runs forward.

STOP

::= STOP

STOP halts execution of a program, terminates any executing text file, returns BASIC to immediate execution, resets the output file to .CONSOLE, and redisplays the prompt character. STOP displays a message, for example

```
?BREAK IN 8712
```

where 8712 is the line number of the program line containing the STOP statement. The program in memory is not altered in any way. STOP has no options associated with it.

STR$

::= STR$(aexpr)

```
)PRINT STR$(25/3)
8.33333
)PRINT STR$(100000000000)+"More"
1E+11More
)
```

STR$ evaluates a given arithmetic expression and returns the value as a string.

Strings

A *string* is a sequence of characters. String variable names must end with a dollar sign ($). Strings may contain from 0 characters (the null string) to 255 characters. The number of characters in a string is referred to as its *length*. Strings are not fixed in length, but may grow or shrink as necessary.

When a program is run, all string variables initially contain the null string.

SUB$

::= SUB$(svar, aexpr [,aexpr]) = sexpr

```
)F$="Hardware" : SUB$(F$,1)="Soft" : PRINT F$
Software
)F$="Hardware" : B$="Soft" : SUB$(F$,1,2)=B$ : PRINT F$
Sordware
)
```

SUB$ lets you replace any part of a string with a specified substring. The string to be changed can be any string variable, and the substring may be the value of any string expression. You must specify the first character in the string to be changed by following that string with an arithmetic expression.

You may optionally include a second arithmetic expression to specify the number of characters in the substring to replace characters in the original string.

SWAP

::= SWAP var1, var2

)SWAP Tick,Tock
)SWAP Old$,New$

SWAP exchanges the value stored in one variable for the value stored in another. You can use string, long integer, regular integer, and real variables with SWAP, but both variables must be of the same type.

TAB

::= TAB(aexpr)

)PRINT "Great"; TAB(8); 347
Great 347
)PRINT "Underhanded"; TAB(8); 553
Underhanded553

TAB is used in PRINT statements to define the number of spaces from the left margin of the text window to begin printing text. If you specify an expression that is less than the number of the current print position, no spaces will be inserted before the next character to be printed.

TAN

::= TAN(aexpr)

```
)PRINT TAN(3.141)
-5.92653E-04
)
```

Returns the tangent of an angle given in radians.

TEN

::= TEN(sexpr)

```
)PRINT TEN("HEXNUM 030C")
)PRINT TEN("CCCC")
```

TEN returns the decimal (base 10) equivalent of the last four characters of the given string expression. The value returned will be in the range -32768 to 32767. The last *four* characters of the value of the given string expression must represent a hexadecimal value; if not, an ?ILLEGAL QUANTITY ERROR results.

TEXT

::= TEXT

The TEXT statement sets the display screen to the usual full-screen text mode, clearing any other text or graphics mode in use, and displays a prompt and the cursor on the next line down at the left margin.

TRACE

::= TRACE

```
)TRACE
)RUN Sammy
```

TRACE prints a "#" followed by the number of each line of a program as it executes. TRACE has no options.

After some study of the results, you would hopefully know what made Sammy run, whether correctly or not. TRACE is switched off by rebooting, LOAD pathname, RUN pathname, or by typing NOTRACE. CHAIN or RUN do not cancel TRACE.

TYP

::= TYP(filenum)

)ON TYP(3) GOSUB 1000,1200,1400,1600,1800,2000

TYP is used to determine what type of data will be read from a particular file on the next access to that file. The argument to the function can be any arithmetic expression, but its value must specify a particular file reference number. The number returned by the TYP function denotes what type of data will next be read from the specified file.

For a data file, TYP returns the following values:

Value	Meaning
0	File type indeterminate
1	Next datum is Real
2	Next datum is Integer
3	Next datum is Long Integer
4	Next datum is String
5	End of file

For a text file, TYP always returns the value 8. If there are no more characters in the file, the value returned is 5.

If the type of a file is as yet undetermined (i.e., whether it is a data or text type) a zero value will be returned for the TYP function.

UNLOCK

See LOCK.

VAL

::= VAL(sexpr)

```
)PRINT 10 * VAL("1.3E4")
)PRINT VAL("13"+"77")
```

VAL evaluates a given string expression and returns the value as a real or an integer number.

If any character of the string expression value evaluated is not a legal numeric character (leading spaces are acceptable), a ?TYPE MISMATCH ERROR occurs.

If the absolute value of the number represented by the value of the string expression is greater than 1E38, an ?OVERFLOW ERROR occurs.

A string expression value containing more than 255 characters causes a ?STRING TOO LONG ERROR.

VARIABLE TYPES

There are four elementary *variable types* in Apple Business BASIC: integers, reals, long integers, and strings. The first three types represent numbers of various kinds, the last type represents sequences of characters.

The *type* of a variable is determined by the last character of its name: % for integer, $ for string, and & for long integer. In the absence of any of these special trailing characters, the variable type is considered to be real by default.

Here are examples of names of the four variable types:

Variable Name	**Variable Type**
Length	real
Marbles7%	integer
Light.Years&	long integer
Myname$	string

VPOS and HPOS

These modifiable reserved variables contain the vertical and horizontal positions, respectively, of the current print position. Changing their values will change the current print position (and the cursor's position). The position of the cursor may be found by accessing the values of VPOS and HPOS.

Assigning values greater than the height of the text window to VPOS, causes the cursor to move to the bottom screen line within the window. Assigning values greater than the width of the text window to HPOS causes the cursor to move to the right margin of the window. The value 0 is converted to the value 1. Assigning values outside the range 0 to 255 to either VPOS or HPOS causes the ?ILLEGAL QUANTITY ERROR message to appear.

WINDOW

::= WINDOW aexpr1, aexpr2 TO aexpr3, aexpr4

WINDOW allows you to set the position and size of the text window, a rectangle within the total screen area where BASIC may display text. For example:

)WINDOW 37,9 TO 44,16

The first pair of numbers specifies the horizontal and vertical coordinates of the upper left corner of the text window, and the second pair specify the coordinates of the lower right corner. The example above will create a text window 8 columns wide and 8 screen lines high, in the center of your screen. When a WINDOW statement is executed, the cursor moves to the lower left corner of the specified window.

WINDOW statement coordinates may be specified by any arithmetic expression. Each of the four expressions must have a value within the range 0 to 255, or an

?ILLEGAL QUANTITY ERROR

message will be displayed. If your values would make the window larger than the maximum allowed screen size, the window is truncated to fit.

WRITE#

::= WRITE# filenum [, recnum] [; expr[{, expr}]]

```
)WRITE# 3; MAJOR%, MINOR%, XLOW
)WRITE# 4, 11; MAP(1,3 ,5,7,9)
```

WRITE# sequentially writes the value of each item in its expression list to a field in a specified data file. You may optionally follow the file number with a comma and an arithmetic expression specifying a record number at which to begin access. The list of expressions must follow the file number (or optional record number), and the expressions in the list must be separated by commas.

One line of data is written for each expression in the list. WRITE# performs no numeric to string type conversions while transferring information from the expressions to the file, it just writes a binary image of numeric data to the file.

If a record number is specified, then the value of the first expression in the expression list is written to the first field in the specified record. Otherwise, records are accessed sequentially.

If there is not enough room left in a record to hold the next value, the field will be written in the next record. Note that writing data to a record causes any old data in the record to be lost. If an attempt is made to write a data field longer than the record length specified when the file was created, the message

```
?OUT OF DATA ERROR
```

is displayed.

BUTTON and PDL

::= BUTTON(0<= aexpr<= 3)
::= PDL(0 <= aexpr <= 3)

Why are these statements here in this chapter? They are the only statements in Business BASIC that require plugging in an external device to obtain some reasonable results. For more information about port A and port B, refer to the *Apple III Owner's Guide.*

BUTTON returns a value depending on the state of the switch specified by its argument. BUTTON(0) and BUTTON(1) are associated with the device connected to port B on the back of your Apple III, and BUTTON(2) and BUTTON(3) specify those connected through port A.

If the switch is closed (button pushed), a value of 255 is returned; otherwise, BUTTON returns 0.

PDL returns a value ranging from 0 to 255, depending on the position of the specified axis of a joystick (or equivalent device) plugged into port A or port B at the back of the Apple III. Some examples follow:

```
10 IF SGN(BUTTON(2)) THEN GOTO 1050 : ELSE GOTO 2300
20 VPOS=(PDL(0)/10) : HPOS=(PDL(1)/30)
```

A

ASCII Character Codes

ASCII is an acronym for American Standard Code for Information Interchange.

The range of standard ASCII codes extends from 0 to 127. Apple Business BASIC also treats the range of values 128 to 255 as valid codes. Certain otherwise unused keystroke combinations are used to represent the additional characters that your Apple III can display (see the chart below).

Legend:

DEC: ASCII code in decimal notation.

HEX: ASCII code in hexadecimal notation.

CHAR: ASCII mnemonics.

CONTROL: Holding down the CONTROL key, while simultaneously pressing any other key generating an ASCII character in the range from decimal 64 to 95 causes the character generated to be equal to the normal character code generated minus decimal 64. Thus, CONTROL-A is 1. Codes below decimal 64 are not affected: ! is a decimal 33 and CONTROL-! is also 33.

Control Characters

DEC	HEX	CHAR	Keyboard Action	Comments and Notes
0	00	Null	CONTROL-@	Null
1	01	SOH	CONTROL-A	
2	02	STX	CONTROL-B	
3	03	ETX	CONTROL-C	Halts execution
4	04	ET	CONTROL-D	
5	05	ENQ	CONTROL-E	
6	06	ACK	CONTROL-F	
7	07	BEL	CONTROL-G	Beeps speaker
8	08	BS	CONTROL-H	Backspace, (same as <--)
9	09	HT	CONTROL-I	Horiz. tab
10	0A	LF	CONTROL-J	Linefeed
11	0B	VT	CONTROL-K	Vert. tab
12	0C	FF	CONTROL-L	Formfeed
13	0D	CR	CONTROL-M	Car. return (same as RETURN)
14	0E	SO	CONTROL-N	
15	0F	SI	CONTROL-O	
16	10	DLE	CONTROL-P	
17	11	DC1	CONTROL-Q	
18	12	DC2	CONTROL-R	
19	13	DC3	CONTROL-S	
20	14	DC4	CONTROL-T	
21	15	NAK	CONTROL-U	
22	16	SYN	CONTROL-V	
23	17	ETB	CONTROL-W	
24	18	CAN	CONTROL-X	Cancels line being edited
25	19	EM	CONTROL-Y	
26	1A	SUB	CONTROL-Z	
27	1B	ESC	ESCAPE	Cursor control and editing
28	1C	FS	CONTROL-SLASH	
29	1D	GS	CONTROL RIGHT BRACKET	
30	1E	RS	CONTROL- ^	
31	1F	US	CONTROL SHIFT UNDERLINE	

See your Apple III Standard Device Drivers manual for additional explanations of how control characters can effect the operation of your Apple.

Uppercase Letters, Numbers, Symbols

DEC	HEX	CHAR	Keyboard
32	20	Space	Spacebar
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?

Uppercase Letters, Numbers, Symbols

DEC	HEX	CHAR	Keyboard
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	[	[
92	5C	\	\
93	5D	]	]
94	5E	^	^
95	5F	—	—

Lowercase Letters and Symbols

DEC	HEX	CHAR	Keyboard
96	60	‘	‘
97	61	a	a
98	62	b	b
99	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q
114	72	r	r
115	73	s	s
116	74	t	t
117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	{	{
124	7C	|	|
125	7D	}	}
126	7E	—	—

B

Errors

When BASIC detects a program error during deferred execution, it checks to see if an ON ERR statement is in effect. If so, program execution jumps to the statement list following the reserved words ON ERR. If not, BASIC halts execution of the program, displays a brief error message, and returns you the prompt and cursor. Variable values and the program text remain intact, but the program can not be continued. All program stacks and FOR/NEXT loop counters are reset to 0.

Errors in a deferred execution statement are not detected until that statement is executed.

Format of Error Messages

When an error occurs in an immediate execution statement, an error message is immediately displayed. For example,

```
)PRINT MID$(234)
?TYPE MISMATCH ERROR
)
```

When an error occurs in a deferred execution statement, an error message is displayed, complete with the line number of the erroneous statement. For example,

```
)10 PRINT MID$(234)
)RUN
```

```
?TYPE MISMATCH ERROR IN 10
)
```

Error Messages

The following is an alphabetical listing of all of the possible BASIC error messages, complete with possible explanations for why the error occurred.

?BAD PATH ERROR

This error message will be displayed if you specify either a pathname of a nonexistent file, or include an illegal character in the pathname.

?BAD SUBSCRIPT ERROR

An attempt was made to reference an array element that is outside the bounds of the array. There are two primary causes of this error:

- attempting to access a nonexistent dimension in an array; for example:

```
)DIM TIMES(23,59)
)TIMES(11,30,27)=Trigger
```

- attempting to access a nonexistent element in any dimension. For example,

```
)DIM Dates(11,30,2000)
)Dates(8,28,2001)=-1
```

?CAN'T CONTINUE ERROR

This error will occur if you are attempting to continue a program after modifying anything other than the variables in the current program.

?DATA WON'T FIT ERROR

An attempt was made to write more bytes of data to a record in a disk file than will fit.

?DEVICE NOT FOUND ERROR

There are three possible causes:

- the device you are referring to is not properly connected;
- your system is not configured for the device you specify;
- the device name you specify includes an illegal character.

?DISK FULL ERROR

There is no space left for additional information on the disk. Either delete files or use another disk.

?DIVISION BY ZERO ERROR

The dividend of any number divided by 0 is infinity. Infinity as such is not illegal, but it does exceed the bounds of memory to store it, which *is* an error.

?DUPLICATE FILE ERROR

An attempt was made to rename a file to a name that already exists on the current disk.

?EXTRA IGNORED

More values were supplied than were asked for by INPUT statement.

?FILE LOCKED ERROR

An attempt was made to modify a locked file.

?FILE NOT FOUND ERROR

An attempt was made to access a file that does not exist on the disk.

?FILE NOT OPEN ERROR

An attempt was made to access a file before opening it with the OPEN statement.

?FILE TOO LARGE ERROR

The file exceeds the size limit for its type as set by SOS.

?FILES BUSY ERROR

This error will occur if you attempt to delete or rename a file while it is open.

?FORMULA TOO COMPLEX ERROR

This error has two possible causes:

- parentheses in an expression are nested more than 14 deep;
- an attempt was made to evaluate an arithmetic expression with more than 14 pending operations caused by precedence.

?ILLEGAL DIRECT ERROR

Given when an INPUT, DEF FN, GET, RESUME, ON ERR, ON KBD, or ON EOF# statements are used in immediate execution; they may only be used in deferred execution.

?ILLEGAL QUANTITY ERROR

The parameter passed to a function was out of range. ILLEGAL QUANTITY errors can be caused by:

- a negative array subscript (for example, A(-1));
- using the LOG function with a negative or zero argument;
- using the SQR function with a negative argument;
- using MID$, LEFT$, RIGHT$, VPOS, HPOS, SPC, WINDOW, TAB, SUB$, CHR$, HEX$, TEN, INSTR, SCALE, or ON...GOTO with an expression whose value lies outside the allowable range;
- opening a file with a record length less than 3;
- specifying a file number less than 1 or greater than 10;

- using a repeat value greater than 255 in a PRINT (#) USING statement or an IMAGE specification;

- a value with an integer range (-32768 to 32767) was expected, but a value beyond that range was encountered.

?INVOKE ERROR

The pathname given in an INVOKE statement specified a non-invokable file.

?I/O ERROR

A physical operation of a peripheral failed. Either a mechanical or electrical problem caused a loss of data: check all external device connections for possible problems. (Is everything plugged in properly?)

?NEXT WITHOUT FOR ERROR

There are three possible causes:

- improper nesting of loops; control variables in a NEXT statement must be listed in the reverse order that they were encountered in FOR statements;

- the control variable specified in a NEXT statement does not correspond to the variable in any FOR statement still in effect;

- a NEXT statement without a specified control variable was executed when no FOR statement was in effect.

?NOT SOS ERROR

The diskette being accessed is not SOS-formatted.

?OUT OF DATA ERROR

There are two possible causes:

- READ statement was executed but all of the data elements in DATA statements in the program have already been read.

- a READ# or WRITE# statement ran out of data when reading from a file; in other words, an end of file was reached.

?OUT OF MEMORY ERROR

There are four possible causes:

- there is no memory available for a file buffer when you open a file.
- program too large.
- an invoked file will not fit in the available memory.
- too much space used by variables.

?OVERFLOW ERROR

The result of a calculation was too large to be represented in BASIC number format. In other words, the absolute value of the number is greater than 1.7E38.

?PATH NOT FOUND ERROR

Part of the pathname specified was invalid: either part of the pathname specified is nonexistent, or an illegal character has been included.

?RANGE ERROR

An illegal line range was specified in a DEL or LIST statement.

?REDIM ERROR

After an array was defined, another DIM statement for the same array was executed. This error often occurs if an array has been given the default dimension 10 because a statement such as A(I)=3 is followed later in the program by a DIM A(100).

?REENTER

Value given for INPUT is of wrong type.

?RETURN WITHOUT GOSUB ERROR

More RETURN statements and/or POP statements were executed than GOSUB statements.

?SOS CALL ERROR

An error occurred within your Apple's operating system, SOS, that is not recognized by BASIC.

?STACK OVERFLOW ERROR

There are three possible causes:

- FOR/NEXT loops nested more than 9 deep;
- subroutines nested more than 23 deep;
- ON KBD subroutines have been entered more than 27 times without a RETURN.

?STRING TOO LONG ERROR

The value of a string expression is greater than 255 characters in length.

?SYNTAX ERROR

Any of the following can cause this error:

- missing parenthesis in an expression;
- illegal character in a statement;
- ON not followed by GOTO or GOSUB;
- IF not followed by THEN or GOTO;
- arithmetic operation on a string;
- a digit as the first character of variable name;
- attempt to use something other than a real for a user defined function;

- variable name over 64 characters in length;
- bad specification in an IMAGE format;
- bad AS option for OPEN#;
- bad operator;
- following DEL with something besides a digit;
- anything else that is not syntactically correct.

?TYPE MISMATCH ERROR

Any of the following can cause this error:

- the left hand side of an assignment statement was a numeric variable and the right hand side was a string, or vice versa;
- a function which expected a string argument was given a numeric one, or vice versa;
- exponentiation with long integer values;
- specifying a non-BASIC file when a BASIC file was expected;
- using a non-real variable when a real variable was expected;
- using strings in an IF...THEN statement;
- wrong IMAGE specification for string/numeric in PRINT (#) USING statement;
- READ# numeric data when next data is a string, or vice versa.

?UNDEF'D FUNCTION ERROR

Reference was made to a user-defined function which had never been defined.

?UNDEF'D STATEMENT ERROR

Any of the following can cause this error:

- An attempt was made to GOTO, GOSUB or THEN to a statement line number which does not exist or has been deleted;
- PRINT USING line when line does not exist;
- IMAGE is not the first statement in the line, or the IMAGE list is null.

?VOLUME NOT FOUND ERROR

The volume name specified by an I/O statement does not match the volume name of the disk.

?WRITE PROTECT ERROR

The files on the diskette cannot be modified because the disk's write-protect notch is covered.

ERROR CODES

When BASIC detects an error, the reserved variable ERR will contain a number code corresponding to the following table.

Error Code Table

Error Code	Error Description
1	NEXT WITHOUT FOR
2	SYNTAX
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	ILLEGAL QUANTITY
6	OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED STATEMENT
9	BAD SUBSCRIPT

10	RANGE
11	INVOKE
12	STACK OVERFLOW
13	REDIM'D ARRAY
14	DIVISION BY ZERO
15	ILLEGAL DIRECT
16	TYPE MISMATCH
17	STRING TOO LONG
18	FORMULA TOO COMPLEX
19	CAN'T CONTINUE
20	UNDEF'D FUNCTION
22	SOS CALL
23	FILES BUSY
24	NOT SOS
25	I/O
26	FILE TOO LARGE
27	WRITE PROTECT
29	BAD PATH
30	FILE NOT FOUND
31	PATH NOT FOUND
32	VOLUME NOT FOUND
33	DUPLICATE FILE
34	DISK FULL
35	FILE LOCKED
36	FILE NOT OPEN
37	DEVICE NOT FOUND
253	EXTRA IGNORED
254	REENTER
255	BREAK (CONTROL-C)

C

Alphabetical List of Reserved Words

The following is an alphabetical list of the reserved words in Apple Business BASIC. Note that some must end with a left parentheses to be considered reserved words. For example, AND is an illegal variable name, but ABS is not.

ABS(
AND
AS
ASC(
ATN(
BUTTON(
CAT
CATALOG
CHAIN
CHR$(
CLEAR
CLOSE
CONT
CONV(
CONV$(
CONV%(
CONV&(
COS(
DATA
DEF
DEL
DELETE
DIM
DIV
ELSE
END
EOF
ERR
ERRLIN
EXFN.
EXFN%.
EXP(
EXTENSION
FN
FOR
FRE
GET
GOSUB
GOTO
HEX$(
HOME
HPOS
IF
IMAGE

INPUT
INSTR(
INT(
INVERSE
INVOKE
KBD
LEFT$(
LEN(
LET
LIST
LOAD
LOCK
LOG(
MID$(
MOD
NEW
NEXT
NORMAL
NOT
NOTRACE
OFF
ON
OPEN
OR
OUTPUT
PDL(
PERFORM
POP
PREFIX$
PRINT
READ
REC(
REM
RENAME
RESTORE
RESUME
RETURN
RIGHT$(
RND(
RUN
SAVE
SCALE(
SGN(
SIN(
SPC(
SQR(
STEP
STOP
STR$(
SUB$(
SWAP
TAB(
TAN(
TEN(
TEXT
THEN
TO
TRACE
TYP(
UNLOCK
USING
VAL(
VPOS
WINDOW
WRITE

D

Variable Maps

Simple Variable Format

Every simple variable in Apple Business BASIC has an entry in memory with the following format:

LENGTH	NAME	TYPE	VALUE

LENGTH is a one byte field that contains the size of the entire variable entry in bytes.

NAME is a field of variable length that contains the ASCII code of the simple variable name. NAME is between 1 and 64 bytes in length.

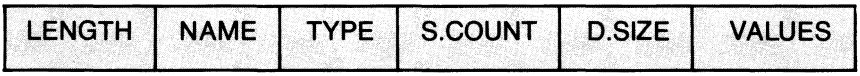
TYPE is a one byte field that contains a code for the type of the simple variable. For reals TYPE=0, for long integers TYPE=$40, for integers TYPE=$80, and for strings TYPE=$FF.

VALUE is a field which contains the value of the variable. The length and contents of the VALUE field depend upon the variable type. Two-byte fields contain the high-order byte first and the low-order byte second:

Type	Byte	Contents
Integer	0	high byte
	1	low byte
Real	0	exponent/sign
	1-4	mantissa
String	0	length
	1	high byte of location
	2	low byte of location
Long integer	0	low byte
	7	high byte

Array Variable

Every array variable in BASIC has an entry in memory with the following format:



LENGTH is a two byte field that contains the size of the entire array variable entry in bytes.

NAME is a field of variable length that contains the ASCII characters of the array variable name. NAME is between 1 and 64 bytes in length.

TYPE is a one byte field that contains a code for the type of the simple variable. For reals TYPE=0, for long integers TYPE=$40, for integers TYPE=$80, and for strings TYPE=$FF.

S.COUNT is a one byte field that contains the number of subscripts in the array variable.

D.SIZE is a field that contains the size of each dimension in the array. Its length, in bytes, is equal to twice the number of dimensions.

VALUES is a field containing the values of each of the array elements. The array elements are stored with the rightmost index ascending slowest. The length and contents of the VALUES field depend upon the variable:

Type	Byte	Contents
Integer	0	high byte
	1	low byte
Real	0	exponent/sign
	1-4	mantissa
String	0	length
	1	high byte of location
	2	low byte of location
Long integer	0	low byte
	7	high byte

E

Space Savers

In order to make your program fit into less memory space, the following hints may be useful.

1. Use integer instead of real arrays wherever possible (see STORAGE ALLOCATION INFORMATION later in this appendix, and the appendix VARIABLE MAPS).

2. Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
)10 P=3.14159
```

in the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

3. The END statement is strictly optional. You can save a byte or two by omitting it from your programs. Don't forget to use END to prevent your program from 'crashing' into its own subroutines, though.

4. Re-use the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use T again. Or, if you are asking the computer user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable to store the reply.

5. Use GOSUB statements to execute groups of program statements that perform identical actions.

6. Use the zero elements of arrays; for instance, A(0), B(0,X).

7. When A$="Cat" is reassigned to A$="Dog" the old string "Cat" is not erased from memory. Using a statement of the form

```
X = FRE
```

periodically within your program will cause BASIC to reorganize string storage so that memory space is used more efficiently.

In serious cases only, try the following tricks. But remember that these will make your program harder for someone else to read, harder to debug, and generally less desirable...but you will save memory!

8. Use multiple statements per line. There is a small amount of overhead (4 bytes) associated with each line in the program.

9. Remove all the REM statement lines from your program.

If you do either of the above, you should consider keeping a copy of the program that has REM statements and single-statement lines.

Storage Allocation Information

When a new function is defined by a DEF statement, 6 bytes are used to store the pointer to the definition. Reserved words such as FOR, GOTO, or NOT, and the names of the intrinsic functions such as COS, INT, and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space on the stack is used as follows:

- Each active FOR...NEXT loop uses at least 16 bytes.
- Each active GOSUB (one that has not RETURNed yet) uses at least 6 bytes.
- Each set of parentheses used in an expression uses 4 bytes, and each temporary result calculated in an expression uses 12 bytes.

See the appendix VARIABLE MAPS for an explanation of memory usage by variables and constants.

F

Memory Usage

This appendix discusses how much memory space is used by Apple Business BASIC to store variables and constants. If you are a beginning programmer, you might skip on, because the information given here is not essential to learning how to program.

A *byte* is the smallest individually accessible unit of memory, and each byte contains eight binary digits or bits. A 128K Apple III has 131072 bytes of memory. After BASIC and SOS have taken their space, there is about 60K of memory available for you to use, depending on the number and size of the drivers in your system. You can find out how much space is available to you by typing

```
PRINT FRE
```

Constants

All constants require 1 byte of memory per digit, including "." and "E" for reals. For instance, the constant

```
2.71828182846
```

uses 13 bytes.

Numeric Variables

One byte of memory is used for each character of a variable name (up to a maximum of 64 bytes).

Each real variable uses 4 bytes of memory to store a value (3 bytes for the mantissa, and 1 for the exponent). For example, the statement

```
)Parish=4977623
```

uses 12 bytes: 6 for the variable name, 4 for the value, and one each for type and link bytes. The type byte tells BASIC what the type of the variable being stored is, and the link byte tells BASIC where the next variable in memory begins.

Each integer variable uses 2 bytes of memory for the value.

Each long integer variable uses 8 bytes of memory for the value.

Strings

Apple Business BASIC stores strings in two parts. The first part is the *string descriptor*, and the second part contains the actual sequence of characters in the string. The string descriptor is 3 bytes long: the first byte of the string descriptor contains the string length, and the last two bytes contain a pointer to the memory location of the first character in the string.

In addition to the three descriptor bytes, each string variable uses one byte for each character of the string name (except the final dollar sign), 1 link byte, and 1 type byte, plus 1 byte for each character in the string. An additional 3 bytes are needed for overhead, unless the value of the variable is the null string. For example, the statement

```
)TR$ = ""
```

uses 7 bytes: 3 for the string descriptor, 2 for the string name, 1 for the link, and 1 for the variable type. The statement

```
)STAR$ = "WIX"
```

uses fifteen bytes: 3 for the string descriptor, 4 for the string name, 1 for the link, 1 for the variable type character, 3 for the string characters, plus 3 overhead bytes.

Apple Business BASIC reserves part of memory as space to be used solely by strings. Exactly how much memory is available for string storage depends on how many other variables and arrays exist in a program and the size of the program itself. When a string decreases in length, BASIC does not immediately reclaim the freed memory space. Instead, whenever BASIC sees that it is about to run out of usable memory, it reorganizes string storage so that memory is used more efficiently. All of the freed string storage space is then reused by new strings.

> You can cause BASIC to reorganize memory space by executing a FRE statement.

Arrays

Each array requires the following amount of memory: 2 link bytes, 1 type byte, 1 byte per character of the array name, 1 byte recording the number of array dimensions, and 2 bytes per dimension.

Each integer array element occupies 2 bytes of memory, real array elements occupy 4 bytes, long integer array elements occupy eight bytes, and string array elements occupy 3 bytes (the string descriptors). For example, the statement

```
)DIM Paymt%(9,3,5)
```

in addition to the 480 bytes taken up by the array that it generates, uses 15 bytes: 2 link bytes, 1 type byte, 5 bytes for the array name, 1 byte for the number of dimensions, and 6 bytes for the 3 dimensions.

> For further information about memory usage, see the appendices VARIABLE MAPS and SPACE SAVERS.

G

Speeding Program Execution

These hints should improve the execution speed of your Apple Business BASIC programs. Note that some of these hints are the same as those used to decrease the memory space used by your programs. This means that in many cases you can increase the speed of your programs at the same time you improve the efficiency of their memory use.

1. This is the most important accelerator of all: Use variables instead of constants. It takes more time to convert a constant to its floating point (real number) representation than it does to fetch the value of a simple or array variable. This is especially important within FOR/NEXT loops or other code that is executed repeatedly.

2. Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. For example, this means that a statement such as

```
5 A=0 : B=0 : C=0
```

will place A first, B second, and C third in the variable table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the variable table to find A, two entries to find B and three entries to find C.

3. Use NEXT statements without a specified control variable. NEXT is slightly faster than NEXT A because no check is made to see if the control variable specified in the NEXT is the same as the control variable in the most recently executed FOR statement.

4. When BASIC branches to a new line number, one of two things happens depending on whether the line number is lower than the currently executing line, or higher. If it is a lower number, such as:

```
)1001 GOTO 1000
```

BASIC scans the entire program starting at the lowest line until it finds the referenced line number (1000, in this example). If the new line number is greater than the current line, BASIC only has to search forward from the current line. Here's an example:

```
)1001 GOTO 2048
```

5. You can make your program run faster if you can put often repeated groups of statements (such as subroutines) at the beginning of a program.

6. FOR/NEXT loops execute faster than loops constructed with GOTO statements.

H

Summary of Apple Business BASIC

Syntax Notation

The method used in this Appendix to describe Apple Business BASIC syntax is almost a simple language in itself. After you get used to it, it will speed your understanding of what is syntactically correct.

The syntax of a language is a body of rules that defines the various language elements and how they may be combined. There are simple elements that are combined into compound elements, which in turn can be combined into expressions and statements.

An element is defined like this:

(element to be defined) ::= (some combination of previously defined elements).

Any uppercase letters or punctuation marks appearing on the right side of the definition must be typed exactly as shown. Lowercase letters represent free information that you must fill in. For example, in the definition

goto statement ::= GOTO linenum

the letters "GOTO" must be typed just as shown, followed by any legal line number.

Some definitions have two or more lines containing ::= in them. These lines are equivalent definitions for a given key word.

To combine elements, the following symbols are used. Note that you do not type them when you are entering a program! They are for purposes of describing syntax only!

|	separate alternative elements.
[ ]	enclose optional elements.
{ }	enclose repeatable elements that must occur at least once.
\ \	enclose elements whose values are to be used.
~	indicates that adjacent elements must be separated by delimiters. Delimiters are defined later.

Here's an intuitive example of how this system is used to describe the various parts of BASIC's syntax. In this example, we'll use houses, as they are familiar places to most of us:

house
::= roof{door}{window} [fireplace][all-electric kitchen]

A house has a roof, one or more doors, one or more windows, and may have a fireplace and an all-electric kitchen.

home
::= house|cottage|mansion

A home can be a house, cottage, or mansion.

price
::= \house\

The selling price is the value of the house.

suburban neighborhood
::= house{ house}

Suburban neighborhoods have space between the houses.

urban neighborhood
::= house{house}

Urban neighborhoods have no space between houses.

This notational scheme is known as modified BNF (Backus-Naur Form) after John Backus and Peter Naur, who first adapted it for use with computer languages. Its essential features were invented in ancient times by the Hindu scholar Panini, who was doing research into the structure of Sanskrit.

The first half of this appendix is a list of all elements used in BASIC statements. The second half of the Appendix is an alphabetical listing of all BASIC statements.

Elements

Discrete Elements

letter

::=A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

digit

::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

line number (linenum)

::= {digit} (between 0 and 63999)

special

::= ! | # | $ | % | & | ' | (|) | * | : | = | @ | + | ; | / | > | ? | / | > | [| < | , |] | ^ | . | - | | |

character

::= letter|digit|special

alphanumeric character

::= letter|digit

control character
::= A character generated by pressing a key while holding down the CONTROL key. For example: CONTROL-G

prompt character
::=)

return
::= A press of the key marked RETURN.

Operands

name
::= letter[{letter|digit|.}]

subscript
::= (arithmetic expression [{,arithmetic expression}])

variable name
::= name[%|$|&] [subscript]

array name
::= variable name [subscript]

integer
::= [+|-]{digit}

integer variable name
::= name%

integer variable (ivar)
::= name% [subscript]

real number

::=[+|-]{digit} [.{digit}][E[+|-] digit[digit]]

::=[+|-][digit]. [{digit}][E[+|-] digit[digit]]

real variable name
::= name

real variable
    ::= name [subscript]

long integer
    ::= [+|-]{digit}

long-integer variable name
    ::= name&

long-integer variable
    ::= name& [subscript]

control variable
    ::= real variable|integer variable

literal
    ::= [{character}]

string
    ::= "literal"

null string
    ::= ""

string variable name
    ::= name$

string variable (svar)
    ::= name$ [aexpr]

array variable name
    ::= name [%|$|&](aexpr{,aexpr})

arithmetic variable (avar)
    ::= integer variable|real variable

long-integer arithmetic variable (liavar)
    ::= name& [subscript]

variable (var)
    ::= avar|svar|liavar

varible list (varlist)
::= var [{,var}]

reserved variable (resvar)
::= ERR|KBD|EOF|ERRLIN|HPOS |VPOS|FRE|PREFIX$|INDENT|OUTREC

modifiable resvar
::= HPOS|VPOS|PREFIX$|INDENT|OUTREC

Operators

arithmetic unary operator (auop)
::= +|-

arithmetic binary operator (abop)
::= +|-|*|/| ^

arithmetic operator (aop)
::= abop|auop

long-integer arithmetic operator (liaop)
::= +|-|*|/|MOD|DIV

unary logical operator (ulop)
::= NOT

binary logical operator (blop)
::= AND|OR|=
|<|>|<>| ><|>=|=> |<=|=< |<=>

logical operator (lop)
::= blop|ulop

operator (op)
::= aop|lop

long integer operator (liop)
::= liaop|lop

long operator (sop)
::= +

string logical operator (slop)
::= lop

Expressions

string expression (sexpr)
::= svar|string
::= sexpr sop sexpr

arithmetic expression (aexpr)
::= avar|real|integer|resvar
::= auop aexpr
::= ulop aexpr
::= aexpr op aexpr
::= sexpr slop sexpr

long-integer arithmetic expression (liexpr)
::= liavar|long integer
::= auop liexpr
::= ulop liexpr
::= liexpr liop liexpr

expression (expr)
::= aexpr|sexpr|liexpr

logic expression (lexpr)
::= aexpr|liexpr

Additional Elements

filenum
::= expr

pathname
::= devname|volname[{/subdir}][/ fname]

functionname
::= real variable name

recnum

::= aexpr

recsize

::= aexpr

spec

::= string spec | literal spec | fixspec | scispec | engrspec

rpt (repeat factor)

::= a whole number from 1 to 255

string spec

::= {[rpt] A|C|R}

A reserves a character position in a left-justified string. C reserves a character position in a centered string. R reserves a character position in a right-justified string.

literal spec

::= {[rpt] X|/|string}

An X means print a space. A / character means print a return. A string must be in quotes. When a repeat factor is placed in front of a string, it affects the entire string.

digitspec

::= [rpt] [{#|Z|&}] . [rpt] {#|Z|&}

::= [rpt] {#|Z|&} [.]

reserves one numeric digit position; leading zeros are replaced with spaces.

Z reserves one numeric digit position; leading zeros are printed.

& reserves one position for a numeric digit or a comma; commas are inserted after every third digit starting at the decimal point and working left. Commas are included in the character count and leading zeros are replaced with spaces. At least five digit positions must be reserved to the left of the decimal point.

fixspec

::= [**] [$] [+|-] digitspec
::= [**] [+|-] [$] digitspec
::= [**] [$] digitspec [+|-]
::= $$[+|-] digitspec
::= $$ digitspec [+|-]
::= [++|--] [$] digitspec

+ reserves a character position for the sign to be printed in.

- reserves a character position for the sign. Sign is printed if negative; otherwise a space is printed.

$ reserves a character position for a dollar sign.

** means print asterisks instead of spaces in unused character positions.

++ reserves the rightmost unused position(s) for the sign (and following dollar sign if any).

-- same as ++ except the sign is replaced by a space if it is positive.

$$ reserves the rightmost unused position(s) for a dollar sign (and following numeric sign if any).

You cannot use $$, ++, or -- if you use Z for the digitspec.

scipart

::= [rpt] #|Z

fracpart

::= .[rpt] [{#|Z}]

exp

::= EEE|EEEE

scispec

:= [+|-] [scipart] [fracpart] exp

engrpart
::= ###|ZZZ

engrspec
:= [+|-] engrpart [fracpart] exp

spec
::= string spec | literal spec | fixspec | scispec | engrspec

deferred statement
::= linenum statement return

immediate statement
::= statement return

statementlist
::= statement [{:statement}]

Statements

ABS function
::= ABS(aexpr)

ASC function
::= ASC(sexpr)

ATN function
::= ATN(aexpr)

BUTTON function
::= BUTTON(aexpr)

CATALOG statement
::= CAT[ALOG]

CHAIN statement
::= CHAIN filename [,linenum]

CHR$ function
::= CHR$(aexpr)

CLEAR statement
::= CLEAR

CLOSE statement
::= CLOSE[# filenum]

CONT statement
::= CONT

CONV function
::= CONV(expr)

CONV& function
::= CONV&(expr)

CONV% function
::= CONV%(expr)

CONV$ function
::= CONV$(expr)

COS function
::= COS(aexpr)

CREATE statement
::= CREATE pathname, CATALOG|TEXT|DATA [,aexpr]

DATA statement
::= DATA [literal|string|real|integer|long integer]
[{, [literal|string|real|integer|long integer]}]

DEF FN statement
::= DEF FN functionname (real variable) = aexpr

DEL statement
::= DEL linenum1 [TO|,|- linenum2]

DELETE statement
::= DELETE pathname

DIM statement
::= DIM array variable name

ELSE statement
::= :ELSE [aexpr|linenum]

END statement
::= END

EXEC statement
::= EXEC pathname

EXFN. statement
::= EXFN. pathname [(lexpr|,@var }]>]

EXFN%. statement
::= EXFN%. pathname [(lexpr| @var[{,lexpr|,@var}])]

EXP function
::= EXP(aexpr)

FOR statement
::= FOR control variable = aexpr1 TO aexpr2 [STEP aexpr3]

FRE statement
::= FRE

GET statement
::= GET var

GOSUB statement
::= GOSUB linenum

GOTO statement
::= GOTO linenum

HEX$ function
::= HEX$(aexpr)

HOME statement
::= HOME

IF...THEN statement
::= IF lexpr THEN linenum|statementlist
[:ELSE linenum|statementlist]

IF...GOTO statement
::= IF lexpr GOTO linenum|statementlist
[:ELSE linenum|statementlist]

IMAGE statement
::= IMAGE spec [{, spec }]

INSTR function
::= INSTR(sexpr, sexpr [,aexpr])

INPUT statement
::= INPUT [string,|;] var{,var}

INPUT# statement
::= INPUT# filenum [, recnum] [; var[{,var}]]

INT function
::= INT(aexpr)

INVERSE statement
::= INVERSE

INVOKE statement
::= INVOKE pathname [{,pathname}]

LEFT$ function
::= LEFT$(sexpr, aexpr)

LEN function
::= LEN(sexpr)

LET statement
::= [LET] var|modifiable resvar = \expression\

LIST statement
::= LIST [linenum1] [TO|,|- [linenum2]]

LOAD statement

::= LOAD filename

LOCK statement

::= LOCK pathname

LOG function

::= LOG(aexpr)

MID$ function

::= MID$(sexpr, aexpr1 [, aexpr2])

NEW statement

::= NEW

NEXT statement

::= NEXT [control variable {,control variable}]

NORMAL statement

::= NORMAL

NOTRACE statement

::= NOTRACE

OFF EOF# statement

::= OFF EOF# filenum

OFF ERR statement

::= OFF ERR

OFF KBD statement

::= OFF KBD

ON EOF# statement

::= ON EOF# filenum statementlist

ON ERR statement

::= ON ERR statementlist

ON...GOSUB statement
::= ON aexpr GOSUB linenum {[,linenum]}

ON...GOTO statement
::= ON aexpr GOTO linenum {[,linenum]}

ON KBD statement
::= ON KBD statementlist

OPEN statement
::= OPEN# filenum [AS INPUT| AS OUTPUT|AS EXTENSION],
pathname [, recsize]

OUTPUT# statement
::= OUTPUT# filenum

PDL function
::= PDL(aexpr)

PERFORM statement
::= PERFORM pathname [(lexpr|@ var[{,lexpr|,@var}])]

POP statement
::= POP

PRINT statement
::= ?|PRINT {[,|;] [expr]} [,|;]

PRINT# statement
::= ?#|PRINT# filenum [, recnum] [; expr [{; expr}] [;]]

PRINT USING statement
::= ?|PRINT USING linenum|string|svar; [expr [{, expr}]] [;]

PRINT# USING statement
::= ?#using|PRINT# filenum [, recnum] USING
linenum|string|svar [; expr[{,expr}]] [;]

READ statement
::= READ var [{,var}]

READ# statement
::= READ# filenum [, recnum] [; var[{,var}]]

REC function
::= REC(filenum)

REM statement
::= REM anything

RENAME statement
::= RENAME pathname1, pathname2

RESTORE statement
::= RESTORE

RESUME statement
::= RESUME

RETURN statement
::= RETURN

RIGHT$ function
::= RIGHT$(sexpr, aexpr)

RND function
::= RND(aexpr)

RUN statement
::= RUN [filename[, linenum]]|[linenum]

SAVE statement
::= SAVE filename

SCALE statement
::= SCALE (varname|aexpr)

SGN function
::= SGN(aexpr)

SIN function
::= SIN(aexpr)

SPC specification
::= SPC(aexpr)

SQR function
::= SQR(aexpr)

STEP clause
::= STEP aexpr

STOP statement
::= STOP

STR$ function
::= STR$(aexpr)

SUB$ statement
::= SUB$(svar, aexpr [,aexpr]) = sexpr

SWAP statement
::= SWAP var1, var2

TAB specification
::= TAB(aexpr)

TAN function
::= TAN(aexpr)

TEN function
::= TEN(sexpr)

TEXT statement
::= TEXT

TRACE statement
::= TRACE

TYP function
::= TYP(filenum)

UNLOCK statements

::= UNLOCK pathname

VAL functions

::= VAL(sexpr)

WINDOW statement

::= WINDOW aexpr1, aexpr2 TO aexpr3, aexpr4

WRITE# statement

::= WRITE# filenum [, recnum] [; expr[{, expr}]]

I

Using the Graphics Invokable Module

I

Using the Graphics Invokable Module

The invokable module BGRAF.INV is a library of assembly language routines that provide a convenient BASIC interface to the system's graphics driver, .GRAFIX . Complete details on the .GRAFIX driver are given in the Standard Device Drivers manual.

If you want to do graphics, you must make sure that SOS.DRIVER includes the .GRAFIX driver. If your Apple III was not supplied that way, you must use the System Configuration Program to incorporate the .GRAFIX driver into your operating system. See the Standard Device Drivers manual for information about configuring the .GRAFIX driver into your system.

To start using the .GRAFIX driver, you should first open the file containing that driver. You or your program can do this with an OPEN# statement that uses the driver's local filename. For example:

```
100 OPEN#1, ".GRAFIX"
```

This statement opens the driver .GRAFIX and assigns it the file reference number 1 (you can assign any integer from 1 through 10 that is not already assigned to another file). Remember this file reference number; you will use it when you print text on the graphics screen with PRINT# statements.

Then, when you are ready to use the facilities of the BGRAF.INV module, you or your program must issue an INVOKE statement containing the module's local filename. For example, your program might contain this line:

```
110 INVOKE "BGRAF.INV"
```

When this statement is executed, the routines in the BGRAF.INV module are read into the Apple from disk, and are then available to the program. You can issue the BASIC statements OPEN#1, ".GRAPHICS" and INVOKE "BGRAF.INV" in immediate execution or at any point in a program. In immediate execution, the quotes surrounding a pathname are optional, but they are required in deferred execution program lines.

If you specify an argument value greater than the maximum correct value for any particular argument, BGRAF.INV substitutes the maximum correct value, instead. Similarly, if you specify an argument value less than the minimum correct value for that argument, BGRAF.INV substitutes the minimum correct value. Invoked functions are executed with BASIC's EXFN%. statement. For example,

```
220 C% = EXFN%.XYCOLOR
```

uses BGRAF.INV's function XYCOLOR.

All of BGRAF.INV's functions are designed to return integer values, so the integer form of BASIC's external function statement (EXFN%.) should always be used.

Before proceeding to descriptions of the actual routines in the BGRAF.INV module, we will first present an overview of the general concepts and operations involved. More details about many of these points can be found in the *Standard Device Drivers* manual.

Overview of the Graphics Display

An Apple III graphics display can be thought of as a rectangular array of colored dots. An x,y coordinate system is superimposed on this array to specify the horizontal and vertical position of each dot. The origin (x=0,y=0) is at the lower left-hand corner of the display, with x-coordinates increasing to the right and y-coordinates increasing toward the top of the display. This is strictly an integer coordinate system.

Another feature of the display is an invisible cursor that is used as a position reference in certain operations. Many operations move the cursor, sometimes without affecting the display.

Graphics Modes

There are four different "modes" for displaying Apple III graphics. Each mode is characterized by the number of dots that make up the display and by the colors that are available:

- Mode 0 : This is a black and white mode in which the full-screen display is 280 dots wide by 192 dots high. That is, x-coordinates are in the range from 0 through 279 and y-coordinates are in the range from 0 through 191.

- Mode 1 : In this mode, sixteen colors are available; but there are special limitations described in the section GRAPHICS IN DISPLAY MODE 1, later in this document. Like mode 0, the full-screen display is 280 dots wide by 192 dots high. That is, x-coordinates are in the range from 0 through 279 and Y coordinates are in the range from 0 through 191.

- Mode 2 : This is a black and white mode that offers twice the horizontal resolution of mode 0: the full-screen display is 560 dots wide by 192 dots high. X-coordinates are in the range from 0 through 559 and y-coordinates are in the range from 0 through 191.

- Mode 3 : In this mode, sixteen colors are available at every dot, with no limitations. The full-screen display is 140 dots wide by 192 dots high. That is, x-coordinates are in the range from 0 through 139 and y-coordinates are in the range from 0 through 191.

Note that the full-screen display is 192 dots high in any mode; only the horizontal resolution and color selection vary from mode to mode.

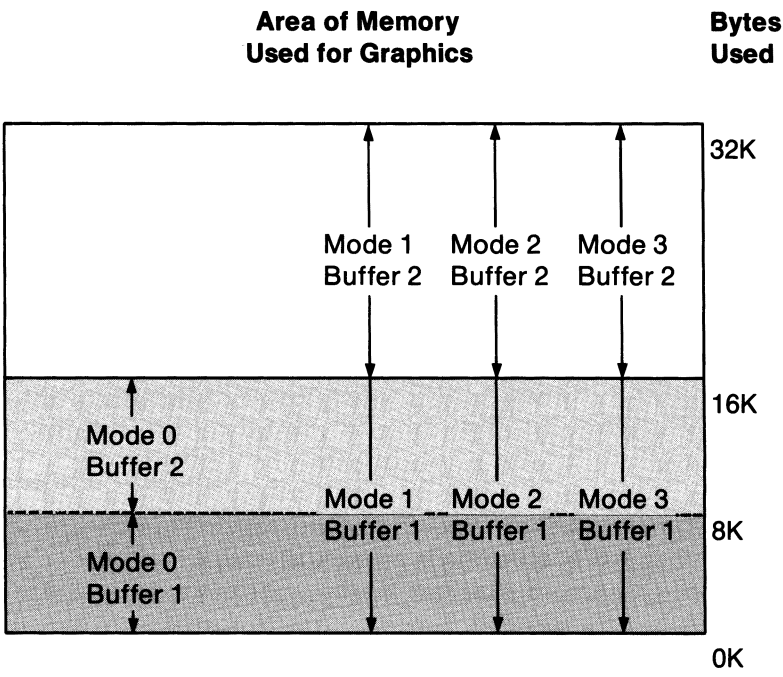
Display Buffers

Most of the graphics routines do not directly affect what appears on the screen. Instead, they affect the current "display buffer". A display buffer is an area in the Apple's memory containing a coded representation of the colors selected for all the dots that make up a display. The graphics routines affect the information in the current display buffer, but that information is not actually shown on the screen until you or your program specifically order this to happen.

Depending on the graphics mode and on the memory size of your Apple III, more than one display buffer may be available simultaneously. This means that your program can show the contents of one display buffer on the screen, and continue to show that image while creating another image in a second display buffer. When the new image is ready, your program can tell the screen to start showing the second buffer's information.

Memory Usage

For each of the four graphics modes, you can select either of two display buffers: buffer 1 and buffer 2. However, this does not mean that there are eight separate, independently available display buffers. For graphics mode 0, each display buffer occupies 8K of memory. For modes 1, 2, and 3, each display buffer occupies 16K of memory. The display buffers are mapped into the Apple's program memory space as follows:



In a 128K Apple III, the entire 32K display buffer space shown is available for graphics. For any given graphics mode, the two display buffers occupy separate areas of memory, so that you can switch back and forth between their independent images. Also, both of the mode 0 display buffers occupy separate areas of memory from display buffer 2 of any other mode. This means you can switch between independent images in mode 0's buffer 2, for example, and mode 3's buffer 2.

Note that buffer 1 of modes 1, 2, and 3 all occupy the same area of memory shared by the two buffers of mode 0. This means, for example, that you cannot switch between independent images in buffer 1 of mode 1 and buffer 1 of mode 2. Also, buffer 2 of modes 1, 2, and 3 all occupy the same area of memory. Thus erasing mode 2's buffer 2, for example, will also effect buffer 2 for modes 1 and 3.

The graphics module automatically reserves enough space in memory for your graphics buffers and takes care of moving any BASIC program out of the graphics memory area. If you later wish to reclaim all or part of the graphics memory space for other program use, BGRAF.INV provides a routine that will do this for you. The BASIC statements LOAD, RUN, and NEW also release all graphics memory for program use.

Overview of the Graphics Routines

This section discusses in general terms what you can do with the routines available after invoking the BGRAF.INV graphics module.

Preparations

When you first open the .GRAFIX driver, several initial default conditions are set which let you immediately begin doing graphics in the primary display buffer for mode 0. A routine is provided that lets you select a different mode and/or a different buffer, if you wish. Another routine tells the Apple to start showing the currently selected graphics display buffer on the screen.

BGRAF.INV automatically reserves enough memory for any display buffer that you use, and moves your BASIC program out of that area of memory.

Colors

The Apple III can provide sixteen colors, identified by the following numbers and names:

Color Number	Color Name	Color Number	Color Name
0	Black	8	Brown
1	Magenta	9	Orange
2	Dark Blue	10	Grey 2
3	Purple	11	Pink
4	Dark Green	12	Green
5	Grey 1	13	Yellow
6	Medium Blue	14	Aqua
7	Light Blue	15	White

In the color graphics display modes (modes 1 and 3), these sixteen colors are sent to the Apple III's COLOR VIDEO output and simultaneously produce a 16-level grey scale at the B/W VIDEO output. In the black and white graphics display modes (modes 0 and 2), all colors other than black are converted to white, at either output.

Control of Color

You can select one color as the "pen color" used for plotting lines and dots, and another color as the "fill color" used for backgrounds and erasing. The BGRAF.INV module provides procedures for selecting these colors.

In the simplest way of using graphics (discussed below), plotting over any display dot changes its color to the current pen color. Similarly, when a filling operation erases any area of the display, this changes all the dots in the area to the fill color.

A great deal can be done with just these simple techniques. More powerful techniques make use of two controllable processes that can modify the way plotting and filling operations affect the display:

- A "color table" can specify the display color that results from applying any source color (pen color or fill color) over any previously existing display color. By default, the color table specifies that the result color is always the same as the source color, but you can change this. This allows you, for example, to draw a green line that appears to go "under" a set of orange lines, or to change a red background to blue without changing the yellow foreground.

- A "transfer option" can further determine how the color resulting from the color table is applied to the actual dot on the display. The effect on the display depends on the color from the color table, and may also depend on the previously existing color of the dot. By default, the transfer option specifies that the display dot always takes on the color resulting from the color table, but you can change this.

Note that, by default, neither the color table nor the transfer option changes the chosen pen color or fill color. In a particular application, only one of these methods of altering the color is normally used. Color graphics modes often use the color table, while black and white modes may use the transfer option more conveniently. However, exotic combinations of both methods may prove useful in certain cases.

The distinction between the two methods is that while the color table works with specific combinations of colors, the transfer option works by performing logical operations on the bit patterns that represent colors internally.

Dots and Lines

The BGRAF.INV module provides a set of procedures for plotting dots and lines (DOTAT, DOTREL, LINETO and LINEREL), or for moving the cursor without plotting (MOVETO and MOVEREL). You plot a dot, or move the cursor to an existing dot, by giving that dot's x,y coordinates. Alternately, you can give x and y displacements instead of absolute coordinates; the displacements are taken relative to the current cursor position.

A line can be plotted (with LINETO) by giving one pair of x,y coordinates; the result is a line from the current cursor position to the specified coordinates. You can also (with LINEREL) give the line's endpoint in terms of x and y displacements from the current cursor position.

Viewports and Areas

One of the BGRAF.INV procedures (VIEWPORT) allows you to define the boundaries of the current "viewport". This is the area of the display that can be affected by plotting and filling operations. By default, the viewport is the whole display, but you can change the viewport to any smaller rectangular portion of the display. If the program tries to plot or erase outside the viewport there is no effect. If a line is plotted and any portion of it is outside the viewport, only the part that is in the viewport actually affects the display.

The filling operation (FILLPORT), used to paint larger areas or to erase images from the display, fills the current viewport with the fill color. Any rectangular area can be quickly colored by specifying an appropriate viewport and then filling it with the chosen fill color.

Text on a Graphics Display

After using a BASIC file-opening statement such as

```
OPEN #1, ".GRAFIX"
```

to assign a file reference number to the .GRAFIX driver (any integer from 1 through 10 may be used), your program can use the same file reference number in PRINT# statements such as

```
PRINT #1; "This is my aunt Merganser."
```

to put text characters into a graphics-mode display. You can use the same system character font used for the normal text display, or you can create and display a new font. BGRAF.INV does not help you to create a new character font (and this can be a difficult task) but it does let you easily display such a font. For more information on fonts, see Creating and Storing a Bit Array.

Copying an Image

A program can use internal data (typically stored as an integer array) to represent a figure, letter, or other image. A specialized procedure is provided that uses the pattern of bits in the array for plotting a pattern of dots on the display. This is a high-speed procedure and is useful for doing animation. BGRAF.INV does not help you create the stored bit pattern that defines the image. For more information on fonts, see the section Creating and Storing a Bit Array.

Saving a Display

Once you or your program have created a graphics display, you can use a BGRAF.INV procedure to save the display onto diskette. Later, another procedure can read the saved information back into the Apple III to re-display the saved image.

Reclaiming Gaphics Memory

If you want to release all or part of the graphics display memory while a program is running, a BGRAF.INV procedure named RELEASE, described later, will help you do this.

Details of the Graphics Routines

Before you can begin using graphics with the Apple III, your operating system must be configured to include the .GRAFIX driver. If your system was not supplied with that configuration, you can use the System Configuration Program to incorporate the .GRAFIX driver into your operating system. See the Standard Device Drivers manual for more information.

To open the .GRAFIX driver, your program can use a BASIC statement such as

```
100 OPEN#1, ".GRAFIX"
```

or you can use the equivalent immediate-execution statement

```
)OPEN#1, ".GRAFIX"
```

These statements assign the file reference number 1 to .GRAFIX, but you could assign any integer from 1 through 10 that is not already assigned to a file. Note that the quotes around a pathname are optional in immediate-execution statements, but they are required for deferred-execution program lines.

To begin using the routines in the invokable graphics module, your program can issue a BASIC statement such as

```
120 INVOKE "BGRAF.INV"
```

or you can issue the immediate-execution statement

```
)INVOKE "BGRAF.INV"
```

Thereafter, if the file BGRAF.INV was successfully loaded from disk, the graphics routines are all available to you and your program. These

routines consist of the following procedures and functions:

- INITGRAFIX to reset four of the default conditions for graphics operations.
- GRAFIXMODE to set the graphics mode and select a display buffer; GRAFIXON to show the current buffer on the screen.
- PENCOLOR and FILLCOLOR to select the colors for plotting and erasing; SETCTAB and XFROPTION to change the color table and transfer option.
- VIEWPORT to set the boundaries of the viewport.
- MOVETO and MOVEREL for moving the cursor; DOTAT, DOTREL, LINETO, and LINEREL for plotting; FILLPORT for erasing the viewport.
- XYCOLOR, XLOC, and YLOC functions to obtain information about the current display.
- NEWFONT and SYSFONT for changing the characters used for text in graphics; DRAWIMAGE for putting a predefined image on the screen.
- GSAVE and GLOAD for saving and retrieving a graphics display.
- RELEASE for making graphics memory space again available for storing and running BASIC programs.

Note that another INVOKE statement first removes any module previously invoked, so that those routines are no longer available.

Invoked procedures are available via the PERFORM statement, and invoked functions are available through the EXFN. statement. You can use a graphics procedure with a BASIC statement such as

```
120 PERFORM GRAFIXMODE (%MODE, %BUFFER)
```

or

```
)PERFORM GRAFIXMODE (%MODE, %BUFFER)
```

Many of the graphics procedures require one or more arguments, which appear in parentheses following the procedure name. Numeric arguments must be passed to these procedures as integers. BASIC will pass an argument as an integer value only if the first character of the argument is a percent sign (%). These are valid numerical arguments:

```
%Q   %MODE   %LEFT%   %32
%HEIGHT%+2.6   %78*3.14   %X/13
```

but these are not valid:

```
Q   MOD   LEFT%   32
HEIGHT%+2.6   78*3.14   X/13
```

If the value of an argument exceeds the limits for integers (-32768 through 32767), an error message is given. Within the limits for integers, if an argument value exceeds the maximum correct value for that particular argument, the maximum correct value is used for that argument, instead. Similarly, if an argument value is less than the minimum correct value for that argument, the minimum correct value is used, instead.

To use a graphics function, a BASIC statement such as this will do:

```
150 HUE% = EXFN%.XYCOLOR
```

or

```
)HUE% = EXFN%.XYCOLOR
```

The graphics functions always return integer values, so you must use the integer form (EXFN%.) of BASIC's external function statement.

The remaining sections are concerned with the actual detailed operation of the procedures and functions of BGRAF.INV.

Preparing for Graphics

The following procedures are all part of getting ready to do graphics. They are normally used before you see any display on the screen.

Opening the .GRAFIX driver normally sets the following initial defaults for graphics:

Graphics Parameter	**Normal Default Set by Opening .GRAFIX**	
Graphics mode	280x192, black & white	(mode 0)
Display buffer	Primary buffer	(buffer 1)
Viewport	Full screen	(x=0 to x=279, y=0 to y=191)
Cursor position	Lower left corner	(x=0, y=0)
Pen color	White	(color=15)
Fill color	Black	(color=0)
Color table	Normal	(no effect)
Transfer option	Normal	(option=0, no effect)
Graphics text font	Current system font	

The "normal" color table and transfer mode specify that the pen color and fill color are placed directly on the screen without alteration during plotting or erasing operations.

If you usually employ a different set of conditions for your graphics, you can change these default conditions using the System Configuration Program, as described in the Standard Device Drivers manual.

Initializing the Display: INITGRAFIX

The INITGRAFIX procedure resets four parameters for the current graphics mode. It has no arguments and can be called at any time. The statement

```
)PERFORM INITGRAFIX
```

sets only these four conditions:

Graphics Parameter	Condition Set by INITGRAFIX	
Viewport	Full Screen	(for currently set graphics mode)
Cursor position	Lower left corner	(x=0, y=0)
Color table	Normal	(no effect)
Transfer option	Normal	(option=0, no effect)

No other graphics parameters are changed by INITGRAFIX . You can use the INITGRAFIX procedure whenever you wish to reset the viewport to the full dimensions of the screen, to move the cursor to the origin, and to reset the color table and transfer option to normal. This can be especially helpful immediately after changing to a different mode or display buffer, or after changing a large number of color table conditions.

Changing Graphics Mode: GRAFIXMODE

To change the graphics display mode or the display buffer or both for future graphics, use the GRAFIXMODE procedure. GRAFIXMODE has two arguments: the first specifies a graphics display mode and the second specifies a display buffer. You must supply both arguments, even if you are changing only one of them. For example, you could use these statements to change to mode 0, secondary display buffer:

```
)MODE = 0 : BUFFER = 2
)PERFORM GRAFIXMODE (%MODE, %BUFFER)
```

After your program performs the GRAFIXMODE procedure, subsequent graphics operations will take place in the selected graphics mode and on the selected display buffer. The effects of this will be visible only after you perform the GRAFIXON procedure, discussed later.

The value of GRAFIXMODE's first argument is an integer from 0 through 3 which selects the display mode for future graphics operations:

Argument Value	Display Mode Selected
0	280 by 192, black & white
1	280 by 192, 16 colors (restricted)
2	560 by 192, black & white
3	140 by 192, 16 colors (no restrictions)

GRAFIXMODE's second argument may have an integer value of either 1 or 2, selecting the display buffer for future graphics operations:

Argument Value	Display Buffer Selected
1	Primary buffer
2	Secondary buffer

See the earlier section OVERVIEW OF THE GRAPHICS DISPLAY for more information about graphics modes and display buffers.

The GRAFIXMODE procedure immediately changes the way your Apple III handles graphics operations, but it does not change the screen display. To make the display on the screen match the mode and buffer selected by a GRAFIXMODE procedure, use the GRAFIXON procedure (discussed in a later section).

The most common use of GRAFIXMODE is to select a graphics display mode, and then to switch back and forth between the two display buffers for that mode. This lets you create an image on a selected buffer before you show that image on the screen. When the image is complete, GRAFIXON will flash it on the screen. And while that image remains on the screen, you can "secretly" select the other buffer for that mode (using GRAFIXMODE) and create a second

image on that buffer. You do not have to show your second image on the screen (using GRAFIXON) until the image is complete.

You can also use GRAFIXMODE (followed by GRAFIXON) to switch rapidly back and forth between two images, to give special effects. Again, the two images must be in display buffers that occupy different areas of graphics memory.

If GRAFIXMODE selects a new mode whose buffer uses the same memory space currently being used for the screen display, subsequent graphics operations may affect the screen strangely. This is because your graphics operations are using the new mode to change the contents of the buffer, but the screen is still interpreting that buffer information according to the mode previously set. Your graphics operations since GRAFIXMODE will be correctly displayed as soon as you perform a GRAFIXON procedure.

Also, an image created in one mode (even a simple blank screen) will rarely make sense when the same buffer information is interpreted in a different mode. For this reason, you will often want to use FILLPORT (see discussion later) to erase the display after you change modes.

For example, to begin doing graphics so that they immediately appear on a cleared screen in mode 3 (primary buffer), you might use these statements:

```
100 MODE = 3 : BUFFER = 1
110 PERFORM GRAFIXMODE (%MODE, %BUFFER)
120 PERFORM FILLPORT
130 PERFORM GRAFIXON
```

Displaying Your Graphics: GRAFIXON

The GRAFIXON procedure causes the currently selected display buffer to be shown on the screen in the currently selected graphics

mode. It has no arguments. When the current graphics mode and/or display buffer are changed with a GRAFIXMODE procedure, all subsequent graphics operations refer to that mode and buffer. However, the mode and buffer displayed on the screen are not changed by a GRAFIXMODE procedure. To change the mode and buffer displayed on the screen to those specified in the most recently performed GRAFIXMODE procedure, use the statement:

```
)PERFORM GRAFIXON
```

The GRAFIXON procedure simply switches the screen display to the currently active mode and display buffer. Note that this is the only way to cause the default or newly selected buffer to appear on the screen. After INVOKE "BGRAF.INV", the screen continues its normal text display. Your program must perform GRAFIXON at some point in order to put any graphics on the screen.

Viewports and Color

The procedures in this section control the "window" through which graphics appear on the screen, and the colors used for drawing and erasing operations. In the most general case, a new dot being plotted must pass through three different "filters" in the following order before the dot is actually placed on the screen:

1. The viewport -- a dot that would be outside the current viewport is ignored.

2. The color table -- the color selected for the dot may be altered by the color table.

3. The transfer option -- after color table change, the dot's resulting color may be further changed by the transfer option.

Under the initial default conditions, all three of these "filters" may be ignored; the viewport occupies the entire screen, and colors are not changed by the color table or the transfer option. Your graphics simply appear on the screen in the place and color you specify.

Setting the Viewport: VIEWPORT

The VIEWPORT procedure sets the boundaries of the graphics viewport. The viewport is a rectangular area of the display that can be affected by subsequent plotting and erasing operations. The VIEWPORT procedure has four arguments, whose integer values give the left, right, bottom, and top coordinates for the viewport you wish to set. For example, to set a viewport that extends horizontally from x=20 to x=100, and vertically from y=40 to y=130, you could use these statements:

```
)LEFT = 20 : RIGHT = 100 : BOTTOM = 40 : TOP = 130
)PERFORM VIEWPORT (%LEFT, %RIGHT, %BOTTOM, %TOP)
```

Once you have set a viewport, dots plotted outside the viewport do not appear on the screen. When you plot lines, letters, and other shapes that would extend beyond the viewport, only the portions within the viewport actually appear on the screen.

To erase the entire viewport to the selected fill color, use the FILLPORT procedure discussed later. An easy way to create a rectangular frame on the screen is to set a viewport, fill it with a frame color, then change to a slightly smaller viewport and fill it with another background color.

The vertical limits of the screen are identical in every mode, extending from y=0 at the bottom to y=191 at the top. The horizontal screen limits vary from mode to mode.

When you change to a new graphics mode, the viewport is automatically adjusted to the full screen dimensions of the new mode.

To reset the viewport to the full dimensions of the screen, you can perform either the INITGRAFIX or VIEWPORT procedures.

Setting the Pen Color: PENCOLOR

The PENCOLOR procedure sets the "pen color", that is the color to be used by subsequent plotting, drawing, and foreground operations. It has one argument, whose integer value specifies the chosen pen color from the following list of colors:

Argument Value	Color Selected	Argument Value	Color Selected
0	Black	8	Brown
1	Magenta	9	Orange
2	Dark Blue	10	Grey 2
3	Purple	11	Pink
4	Dark Green	12	Green
5	Grey 1	13	Yellow
6	Medium Blue	14	Aqua
7	Light Blue	15	White

In black and white graphics modes (modes 0 and 2), all non-black colors appear as white both at the COLOR VIDEO output and at the B/W VIDEO output. In the color modes (modes 1 and 3), the sixteen colors are sent to the COLOR VIDEO output and are simultaneously sent to the B/W VIDEO output as sixteen gradually whiter levels of grey.

For example, to change the pen color to yellow, you could use this statement:

```
)PERFORM PENCOLOR (%13)
```

After you perform PENCOLOR , the selected pen color is used for plotting dots and lines, as the foreground color for text characters on the graphics screen, and as the foreground color for blocks put on the screen by the DRAWIMAGE procedure.

Before a new dot in the selected pen color is placed on the screen, its color may be modified by the color table and/or by the transfer option.

Setting the Fill Color: FILLCOLOR

The FILLCOLOR procedure sets the "fill color", that is the color to be used by subsequent background and area-filling operations. It has one argument, whose integer value specifies the chosen fill color (see the previous section for a table of argument values and their corresponding colors).

For example, to change the fill color to dark green, you could use the statements:

```
)COLOR% = 4
)PERFORM FILLCOLOR (%COLOR%)
```

After you perform FILLCOLOR , the selected fill color is used for erasing the viewport with the FILLPORT procedure, as the background color for text characters on the graphics screen, and as the background color for blocks put on the screen by the DRAWIMAGE procedure.

Before a new dot in the selected fill color is placed on the screen, its color may be modified by the color table and/or by the transfer option.

Fancier Color Operations: SETCTAB

The SETCTAB procedure sets one of the 256 possible color-mapping conditions in the "color table". The color table specifies the color that results from plotting a new dot of a given "source color" (either pen color or fill color) over a previously existing display dot of a given "screen color".

SETCTAB has three arguments. The integer value of the first argument specifies a source color, that may have been selected either by FILLCOLOR or by PENCOLOR . The integer value of the second argument specifies a screen color, that is the existing color of a dot in the current display buffer (whether that buffer is now on the screen or not). The integer value of the third argument specifies the color that will result if you plot a dot of the given source color at a point whose existing color is the given screen color.

For example, to cause subsequent green lines to show up purple where they are drawn over a brown background, you could use these statements:

```
)SOURCECOLOR = 12 : SCREENCOLOR = 8 : RESULTCOLOR = 3
)PERFORM SETCTAB (%SOURCECOLOR, %SCREENCOLOR,
%RESULTCOLOR)
```

Similary, you could use these statements to make a subsequent dark blue line appear to go "under" any existing orange lines it crosses:

```
)SOURCECOLOR = 2 : SCREENCOLOR = 9 : RESULTCOLOR = 9
)PERFORM SETCTAB (%SOURCECOLOR, %SCREENCOLOR,
%RESULTCOLOR)
```

If you have a yellow design on a light blue background, and you wish to change the background to magenta without changing the yellow design, you could use these statements:

```
)SOURCECOLOR = 1 : SCREENCOLOR = 13 : RESULTCOLOR = 13
)PERFORM SETCTAB (%SOURCECOLOR, %SCREENCOLOR,
%RESULTCOLOR)
)PERFORM FILLCOLOR (%SOURCECOLOR)
)PERFORM FILLPORT
```

The first two statements in this example set the color table so that when a magenta dot is plotted over an existing yellow dot, the result is a yellow dot on the screen. Thus yellow screen dots are not changed when magenta is plotted over them. The next three statements plot magenta dots over the entire viewport. This changes the screen to magenta except where yellow dots previously existed on the screen.

The initial default conditions set the color table to "normal", so that any source color results in the same color, unmodified, being passed on to the display (after being processed by the transfer option), no matter what previous colors exist in the display. Here is a diagram of the "normal" color table:

Source Color	Screen Color															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Black	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Magenta	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
D. Blue	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Purple	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
D. Green	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Grey 1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
M. Blue	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
L. Blue	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Brown	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
Orange	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
Grey 2	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Pink	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
Green	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
Yellow	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
Aqua	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
White	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

In this diagram, the intersection of a source color row and a screen color column contains the number of the result color that will go on the screen if you plot that source color over a dot of that screen color. When you perform the SETCTAB procedure, you are changing just one of the 256 intersection numbers in this color table.

To return the color table to "normal" after you have set any special conditions into it, you can use the INITGRAFIX procedure. That procedure also sets the viewport to full screen, and moves the cursor to the lower left corner.

Note that plotting operations pass the color for a new dot first through the color table and then through the transfer option. It is the result of the color table that is used as the source color for the transfer option.

Fancier Black and White: XFROPTION

The XFROPTION procedure sets the "transfer option". The transfer option is similar to the color table: it determines the color which results from plotting a new dot of any "source color" (from the color table) over a previously existing display dot of any "screen color". While SETCTAB sets a single color which results from one particular pair of source and screen colors, XFROPTION sets a transfer option which operates on all pairs of source and screen colors.

XFROPTION has one argument, whose integer value specifies the transfer option to be used for subsequent plotting operations:

Argument Value	**Transfer Option Selected**	**Result Color, Given a Source Color and a Screen Color**
0	Replace	Sourcecolor
1	Overlay	Sourcecolor OR Screencolor
2	Invert	Sourcecolor XOR Screencolor
3	Erase	(NOT Sourcecolor) AND Screencolor
4	Inverse Replace	NOT Sourcecolor
5	Inverse Overlay	(NOT Sourcecolor) OR Screencolor
6	Inverse Invert	(NOT Sourcecolor) XOR Screencolor
7	Inverse Erase	Sourcecolor AND Screencolor

For example, to begin using the "Invert" transfer option for subsequent plotting operations, you could use these statements:

```
)INVERT% = 2
)PERFORM XFROPTION (%INVERT%)
```

Because the effect of the transfer option is rather complex when applied to 16-color displays, the transfer option is most often used in modes which are restricted to black and white displays. The names of the options really make sense only in black and white, where there are just two colors in use: 0 (black) and 15 (white).

- Replace (transfer option 0). This is the default option. It completely ignores the colors on the screen, and simply copies any source color directly onto the display without modification.

- Overlay (transfer option 1). Useful for putting white text characters or image blocks onto a black background crossed by a few, thin white lines. Does not erase any background around characters or image.

- Invert (transfer option 2). Useful for plotting white lines, text, or image blocks onto a background consisting of large areas of both black and white. The lines or shapes will show up as white where they are on a black background area, and as black where the background is white. If you plot the same figure twice in the same place with this option (even in color modes), the second plot erases the figure, leaving the background unchanged.

- Erase (transfer option 3). Black and white plotting has no effect on a black and white background except that a white line, text character, or image crossing a white background area erases a black image of that line or shape.

- Inverse Replace, Inverse Overlay, and Inverse Reverse (transfer options 4, 5, and 6). These options first convert a white source to black, or a black source to white, and then behave just like Copy, Overlay, and Reverse.

- Inverse Erase (transfer option 7). Black and white plotting has no effect on a black and white background except that a black line, text character, or image appears where it crosses a white background area.

In color modes, it is usually easier to modify colors using SETCTAB and the color table, rather than using a transfer option. If you wish to use the transfer option with colors other than black and white, you may need to read the more technical information in the remainder of this section. Alternatively, you can use the "Color Transfer Tables" in the Graphics Quick Reference section at the back of the Standard Device Drivers manual. In that manual, transfer options 0, 1, 2, and 3 are called "Store", "OR", "XOR", and "BIC", respectively.

The transfer option compares each of the four bits in the source color's code to the corresponding bit in the screen color's code. The results of these four comparisons are the four bits which specify the result color. Four different logical operations, in different combinations as specified for each transfer option, characterize the results of these bit-by-bit comparisons:

- The OR operation yields a 1 if either of the two bits being compared is a 1 or if both are 1; it yields a 0 if both bits are 0.
- The XOR operation yields a 1 if either bit is a 1; but it yields a 0 if both bits are 1 or if both bits are 0.
- The AND operation yields a 1 only if both bits are 1; it yields a 0 if either bit is 0 or if both bits are 0.
- The NOT operation affects only one color code; it converts any 0 bit in that code to a 1, and any 1 bit to a 0.

For example, if you set the transfer option "Erase" (option 3), the operation is defined as

Resultcolor = (NOT Sourcecolor) AND Screencolor

Suppose you now plot a dot of source color Light Blue (binary code 0110) over an existing dot of screen color Yellow (binary code 1101). The "Erase" transfer operation in this case is

Resultcolor = (NOT 0110) AND 1101

NOT changes each 0 to a 1 and each 1 to a 0, so (NOT 0110) becomes 1001. The remaining operation is

Resultcolor = 1001 AND 1101

AND yields a 1 only if corresponding bits are both 1. The first bit of each code is 1, so the first result bit is also 1. The two codes' second bits are 0 and 1, respectively, so the second result bit is 0. The third bits are both 0, giving 0 as the third result bit. Finally, both codes have 1 as the last bit, so the last result bit is 1. Thus the four-bit code for the result color is 1001, which specifies the color Orange for the dot on the display.

Each transfer option specifies a bit-wise logical operation to be performed on the four-bit binary equivalent of the source color number and the four-bit binary equivalent of the existing screen color number at the dot being plotted. The result of this operation is a four-bit number whose decimal equivlent specifies the color of the dot which is actually placed on the display.

The source color number for the dot being plotted may have been selected either by FILLCOLOR or by PENCOLOR, and may have been modified by a condition set in the color table by SETCTAB . The screen color number gives the existing color at that dot in the current display buffer (whether that buffer is now on the screen or not). For your convenience, here are the four-bit binary codes specifying the various colors:

Color Number	4-Bit Code	Color Name
0	0000	Black
1	0001	Magenta
2	0010	Dark Blue
3	0011	Purple
4	0100	Dark Green
5	0101	Grey 1
6	0110	Medium Blue
7	0111	Light Blue

Color Number	4-Bit Code	Color Name
8	1000	Brown
9	1001	Orange
10	1010	Grey 2
11	1011	Pink
12	1100	Green
13	1101	Yellow
14	1110	Aqua
15	1111	White

Note that plotting operations pass the color for a new dot first through the color table and then through the transfer option. It is the result of the color table that is used as the source color for the transfer option.

Moves, Dots, Lines, and Areas

The routines discussed in this section are those you will use to do the actual plotting of your graphics images. Moves, dots, and lines all leave the cursor at the last point moved to or plotted. Filling an area does not move the cursor.

Plotting is the same as a move, except that an image (dot or line) is drawn on the screen at the point of the plot.

The absolute x and y coordinates for a cursor-move, dot, or line may have any integer value from −32768 through 32767, even if those coordinates specify a position beyond the boundaries of the viewport or the boundaries of the screen. However, only those dots or line portions will be plotted which lie within the currently set viewport.

A relative move cannot create an absolute cursor, dot, or line position whose coordinates exceed the range from −32768 through 32767. If a relative move would result in a coordinate which exceeds the minimum or maximum limit, that coordinate is adjusted to the limit exceeded.

The color of any dot actually placed in the display may be changed from the chosen pen color or fill color by the color table and/or by the transfer option.

Moving the Cursor, Absolute: MOVETO

The MOVETO procedure moves the cursor to a specified absolute screen location without plotting. MOVETO has two arguments, whose integer values are the coordinates of the position where the cursor is to be placed. The first argument specifies the absolute horizontal or x-coordinate, and the second argument specifies the absolute vertical or y-coordinate. For example, to move the cursor to the position x=133,y=25 , you could use these statements:

```
)X = 133 : Y = 25
)PERFORM MOVETO (%X, %Y)
```

Moving the Cursor, Relative: MOVREL

The MOVEREL procedure moves the cursor to a specified screen location relative to the current cursor location, without plotting. MOVEREL has two arguments, whose integer values indicate how far to move the cursor from its current position. The first argument specifies the horizontal or x-axis distance to move the cursor, and the second argument specifies the vertical or y-axis distance. For example, to move the cursor 35 units to the right and 74 units down from its current position, you could use these statements:

```
)DX = 35 : DY = −74
)PERFORM MOVEREL (%DX, %DY)
```

Plotting Points, Absolute: DOTAT

The DOTAT procedure moves the cursor to a specified absolute screen location and then plots a dot at that location in the current pen color. DOTAT has two arguments, whose integer values are the coordinates of the position where the dot is to be placed. The first argument specifies the absolute horizontal or x-coordinate, and the second argument specifies the absolute vertical or y-coordinate. For example, to plot an aqua-colored dot at the position x=47,y=139 , you could use these statements:

```
)AQUA% = 14
)PERFORM PENCOLOR (%AQUA%)
)X% = 47 : Y% = 139
)PERFORM DOTAT (%X%, %Y%)
```

or you could use these completely equivalent statements:

```
)PERFORM PENCOLOR (%14)
)PERFORM DOTAT (%47, %139)
```

Plotting Points, Relative: DOTREL

The DOTREL procedure moves the cursor to a specified screen location relative to the current cursor location, and then plots a dot at that location in the current pen color. DOTREL has two arguments, whose integer values indicate how far to move the cursor from its current position, before plotting the dot. The first argument specifies the horizontal or x-axis distance to move the cursor, and the second argument specifies the vertical or y-axis distance. For example, to plot a purple dot 113 units to the left and 14 units up from the current cursor position, you could use these statements:

```
)PURPLE% = 3
)PERFORM PENCOLOR (%PURPLE%)
)DX% = -113 : DY% = 14
)PERFORM DOTREL (%DX%, %DY%)
```

Drawing Lines, Absolute: LINETO

The LINETO procedure moves the cursor to a specified absolute screen location and then draws a line from the old cursor position to the new cursor position, in the current pen color. LINETO has two arguments, whose integer values are the coordinates of the point to which the line will be drawn. The first argument specifies the absolute horizontal or x-coordinate of the end point, and the second argument specifies the absolute vertical or y-coordinate. For example, to plot a pink-colored line from the current cursor position to the position x=863,y=-144 , you could use these statements:

```
)PINK = 11
)PERFORM PENCOLOR (%PINK)
)X = 863 : Y = -144
)PERFORM LINETO (%X, %Y)
```

Drawing Lines, Relative: LINEREL

The LINEREL procedure moves the cursor to a specified screen location relative to the current cursor location, and then draws a line from the old cursor position to the new cursor position, in the current pen color. LINEREL has two arguments, whose integer values indicate how far to move the cursor from its current position, before

drawing a line to the new position. The first argument specifies the horizontal or x-axis distance to move the cursor, and the second argument specifies the vertical or y-axis distance. For example, to draw a medium blue line from the current cursor position to a point 8 units to the right and 58 units up from that position, you could use these statements:

```
)COLOR = 6
)PERFORM PENCOLOR (%COLOR)
)DX = 8 : DY = 58
)PERFORM LINEREL (%DX, %DY)
```

or these equivalent statements:

```
)PERFORM PENCOLOR (%6)
)PERFORM LINEREL (%8, %58)
```

The endpoint for the line is adjusted, if necessary, so that its coordinates are within the range from −32768 through 32767. This takes place before the line is drawn to the adjusted endpoint.

Painting a Rectangle: FILLPORT

The FILLPORT procedure takes no parameters. It fills the viewport with the currently selected fill color. Unless special conditions are set in the color table or the transfer option, FILLPORT erases everything in the viewport. For example, to erase the screen to brown, you might use these statements:

```
)COLOR = 8
)PERFORM FILLCOLOR (%COLOR)
)PERFORM FILLPORT
```

Screen Information Functions

The BGRAF.INV module provides three functions for reading the current location of the cursor and the color of the display dot at that point. Your BASIC program uses these functions through the EXFN%. external function statement.

Other information about the current graphics environment can be obtained, if necessary, by transferring the .GRAFIX driver's "screen status block" to a suitable program buffer. The BASIC graphics module BGRAF.INV does not help you do this: see the Standard Device Drivers manual for more information.

Reading the Screen Color: XYCOLOR

The XYCOLOR function returns an integer value which specifies the color of the display dot at the current cursor position. For example, to discover the dot color at location x=23,y=7 , you could use the statements

```
)HORIZ = 23 : VERT = 7
)PERFORM MOVETO (%HORIZ, %VERT)
)COLOR% = EXFN%.XYCOLOR
```

The first two statements move the cursor to the position x=23,y=7 . If the color of the display dot at that position is Grey 2, the third statement will assign the value 10 to the integer variable COLOR% .

XYCOLOR will return the color of a dot, even though that dot is not within the currently set viewport. If the current position of the cursor is not within the boundaries of the screen, XYCOLOR returns the value −1.

Reading the Cursor Position: XLOC and YLOC

The XLOC function returns an integer value which is the horizontal or x-coordinate of the current cursor position. Similary, the YLOC function returns an integer value which is the vertical or y-coordinate of the current cursor position. For example, to discover the present location of the cursor, you could use these statements:

```
)X% = EXFN%.XLOC : Y% = EXFN%.YLOC
```

If the cursor is at the position x=34,y=−598 , this example will assign the value 34 to the integer variable X% and the value −598 to the integer variable Y%.

XLOC and YLOC will return the position of the cursor, even though the cursor is not within the currently set viewport, or is beyond the limits of the screen. There is no error generated if XLOC and YLOC values are between 32767 and −32768, even if the cursor resulting cursor position is off the screen.

Displaying Text and Other Images

The procedures in this section allow you to put shapes other than dots, lines, and rectangles on the screen.

Text can be put into your graphics for labels, legends, or explanations. You can also change to a different set, or "font", of text characters. You could design these characters to looks like another alphabet, such as Greek or Russian, or you could design almost any arbitrary shape corresponding to different letters, such as a person in different stages of running.

Finally, you can add a predefined shape to your graphics display, in any screen position. BGRAF.INV does not help you create a new set of characters or a predefined shape, but it does make it easy to put such characters or shapes on the screen.

For help with creating character sets or stored images, see the section CREATING AND STORING A BIT ARRAY, later in this chapter.

Putting Text Into Graphics

The BASIC statement PRINT# is used to display text on the graphics screen. Before you can use PRINT# , you must first use the OPEN# statement to assign a BASIC file reference number to the .GRAFIX driver. If you did not do this when you invoked BGRAF.INV, use a statement such as this:

```
)OPEN#1, ".GRAFIX"
```

This statement assigns .GRAFIX the file reference number 1, but you could assign any integer from 1 through 10 as the file reference number. You will want to use a number not currently assigned to another file.

Once OPEN# has assigned a file reference number to the .GRAFIX driver, PRINT# can use the same number to send characters to the graphics display. Just move the cursor to the screen position where you want the text to begin, and then start printing with PRINT# . For example, if you have assigned .GRAFIX the file reference number 1, you could print Apple Computer starting in the middle of the left edge of the graphics screen:

```
)PERFORM MOVETO (%0, %86)
)PRINT#1; "Apple Computer"
```

The original cursor position determines the top left corner of the first character's rectangular "cell". Characters can begin at any dot position; they are not limited to normal text-screen character positions. You can print text at any possible cursor position, even beyond the boundaries of the viewport and the screen. However, only those portions of the text which lie within the currently set viewport are actually plotted on the display.

After each character is put on the screen, the cursor is advanced one character-cell width. The cursor's final position depends on your PRINT# statement: graphics text follows normal text-mode formatting for line feeds, RETURNs, concatenations, and tab fields. PRINT# text does not, however, recognize the right edge of the viewport or screen; a very long line of text just disappears when it reaches the viewport's right edge.

Text characters in graphics are normally taken from the same system character font used for text-mode displays. However, the NEWFONT procedure can change the character font used for graphics to a different font created and stored by you.

The same set of characters is used in every graphics mode, but the characters look somewhat different in different modes. You will find that forty standard characters fit across the width of the screen in graphics modes 0 and 1. Mode 2 accomodates eighty standard characters, and mode 3 will display a maximum line of twenty standard characters.

Note that statements which normally put text on the text display will continue to do so, even though a graphics display is on the screen at the time. When you return to text mode (by issuing BASIC's TEXT statement), the characters you printed on the text display will be waiting for you. For example, the statements

```
)PRINT#1; "Meanwhile, back at the ranch..."
)PRINT "And now, here is chapter 13 of our story."
```

will put the words "Meanwhile, back at the ranch..." starting at the graphics cursor position in the current graphics display buffer. They will also put the words "And now, here is chapter 13 of our story." starting at the text cursor in the text-mode display.

While you will probably print most characters in white on a black background for clarity, you can also print text in colors. The character itself takes its color from the current pen color. Each character's small, rectangular background area takes its color from the current fill color. Before they affect the dots in the display, the color selected for each character-dot and background-dot may be altered by the color table and/or the transfer option.

Many character/background color combinations are not easily readable. See the NTSC Color Compatibility Table in the Standard Device Drivers manual for more details about useful colors for text.

Changing Text Fonts: NEWFONT

When you use PRINT# to display text on the graphics screen, the default set of characters, or "font", is the standard Apple III text font. The NEWFONT procedure lets you switch to another previously-stored font for graphics. NEWFONT requires three integer arguments. The first argument is a pointer to the first element of a dimensioned integer array containing the new font. The second argument specifies the width of the new font's character cell in dots, and the third argument specifies the new character cell's height in dots.

For example, suppose you have stored a new font in integer array variable FONT%. The character cell for this font is 12 dots wide by 7 dots high. To start using this font for subsequent PRINT# graphics text, use these statements:

```
)WIDTH = 12 : HEIGHT = 7
)PERFORM NEWFONT (@FONT%(0,0), %WIDTH, %HEIGHT)
```

After these statements, if the .GRAFIX driver has been assigned the file reference number 1 (by OPEN#1, ".GRAFIX"), a statement such as

```
)PRINT#1; "OK"
```

will put the contents of the new font's 80th and 76th character cells on the graphics display, starting at the cursor position. That is because the standard letters O and K are the 80th and 76th characters in the ASCII coding sequence.

The name of the first element of the array variable containing the new font is preceded by an "at" sign (@) where it appears as an argument. This tells BASIC to get a "pointer" to the array variable's first element, and to pass the pointer to the NEWFONT procedure. The pointer then gives NEWFONT the address in the Apple's memory where the new font can be found.

The NEWFONT procedure does not change the characters printed on the text-mode display; text mode continues to use the standard font.

The NEWFONT procedure's first argument is usually a pointer to an array of integers, or perhaps an array of long integers. The BGRAF.INV module does not provide any aid in creating a new character font; it just lets you use such a font easily. The section CREATING AND STORING A BIT ARRAY, later in this chapter, may help you to make a new font yourself.

Returning to the Normal Font: SYSFONT

The SYSFONT procedure lets you begin using the normal system font for graphics again, after you have changed to a new font by the use of NEWFONT . SYSFONT has no parameters. The statement

```
)PERFORM SYSFONT
```

causes subsequent PRINT# statements to send the graphics screen characters from the system font instead of any new font.

The system font is always used for text-mode displays, even when a new font is being used for graphics. Technical details about the system font can be found in the section CREATING AND STORING A BIT ARRAY, later in this chapter.

Drawing Predefined Shapes: DRAWIMAGE

The DRAWIMAGE procedure draws a predefined shape on the screen. It transfers to the current graphics screen a specified portion of a block of bits, placing them below and to the right of the current cursor position. Each 1-bit in the source block portion is put on the screen using the current pen color; each 0-bit use the current fill color. Both colors may be modified by the color table and the transfer function before they affect the display.

DRAWIMAGE requires six integer arguments. The first argument is a pointer to the first element of a dimensioned integer array containing the source block of bits. The second argument specifies the number of bytes (not bits) in each row of the source block. The third argument specifies the number of bits to skip in each source row before beginning a row transfer. The fourth argument specifies the number of source rows to skip before beginning the transfer process. These two arguments determine the top left corner within the source block of the portion to be transferred. The fifth and sixth arguments specify the bit width and height of the block portion to be transferred to the screen.

For example, suppose you have stored your source block in the two-dimensional integer array variable SOURCE% . This source block consists of twenty rows of bits, each row four bytes long. Now suppose you want to take a portion of this block eight bits wide by five rows high, from the lower right corner of the block, and transfer it to the screen so that the current cursor position determines the top left corner of this image on the screen. That is, the block portion to be transferred will consist of the last eight bits of each of the last five rows in the source block. To accomplish this, DRAWIMAGE must skip the first fifteen source rows entirely, and then transfer all but the first twenty-four bits from each of the remaining rows. These statements should do the job:

```
)ROWBYTES = 4 : XSKIP = 24 : YSKIP = 15 :
  DWIDTH = 8 : DHEIGHT = 5
)PERFORM DRAWIMAGE (@SOURCE%(0,0), %ROWBYTES,
  %XSKIP, %YSKIP, %DWIDTH, %DHEIGHT)
```

The name of the first element of the integer array variable containing the source block is preceded by an "at" sign (@) where it appears as an argument. This tells BASIC to get a "pointer" to the array variable's first element, and to pass the pointer to the DRAWIMAGE procedure. The pointer then gives DRAWIMAGE the address in the Apple's memory where the source block's beginning can be found.

The DRAWIMAGE procedure's first argument is usually a pointer to an array of integers, or perhaps an array of long integers. The BGRAF.INV module does not provide any aid in creating a source block; it just lets you use such a stored image easily. The section CREATING AND STORING A BIT ARRAY, later in this chapter, may help you to make a source block yourself.

Preserving Your Graphics

The procedures in this section let you save a graphics display, once you have created it, as a disk file. Later, you can load the contents of this file back into a graphics display buffer for display or for further work.

Saving a Picture: GSAVE

If you would like to save the contents of a graphics buffer at any time, you can use the GSAVE procedure. GSAVE works similarly to the normal SAVE statement, except that it saves a picture instead of a program. Only one argument is required: a pathname. For example, if you execute the statement

```
)PERFORM GSAVE. "SCREEN.PICTURE"
```

the contents of the current display buffer are stored in the disk file SCREEN.PICTURE . The file is essentially a data file, but it has the type FOTO , not DATA . Along with the actual image information, GSAVE also saves the current graphics mode for use when retrieving the file.

Retrieving a Saved Picture: GLOAD

If you would like to transfer the contents of a stored graphics file to the current graphics buffer, you can use the GLOAD procedure. GLOAD works similarly to the LOAD statement, except that it loads a picture instead of a program. Only one argument is required: a pathname. For example, if you execute the statement

```
)PERFORM GLOAD. "SCREEN.PICTURE"
```

the contents of the file SCREEN.PICTURE will be put into the graphics buffer. The file specified must be of type FOTO . The currently selected buffer number is not changed by GLOAD, but the graphics mode is automatically changed to the graphics mode in effect when the stored image was saved. The new image completely replaces any previous display in that buffer, disregarding the viewport, the color table, and the transfer option.

After Graphics

The graphics procedure and BASIC statements discussed in this section are normally used at the end of a program using graphics, especially if that program is going to chain to another program. This information helps you reclaim graphics memory space for other uses, and return to the text-mode display.

Releasing Graphics Memory: RELEASE

When you perform INITGRAFIX , enough memory to contain the default display buffer (usually 8K bytes) is automatically reserved, and your BASIC program is moved out of that area of memory. Thereafter, anytime a GRAFIXMODE or GLOAD procedure changes the graphics mode or the display buffer, additional space in memory is reserved for the new mode or buffer, if necessary. Any memory used for graphics usually remains unavailable for other use by your BASIC programs until a RELEASE is performed.

If you have finished using some or all of the graphics memory space, and you wish to make that memory again available for storing and running BASIC programs, you can use the RELEASE procedure. Each time you issue the statement

```
)PERFORM RELEASE
```

the graphics buffer space highest in memory is reclaimed for use by BASIC, and your BASIC program is moved back down into that space.

RELEASE has no arguments, and it reclaims memory space in steps. If you were using the maximum amount of graphics display memory (32K bytes), you would need to perform the RELEASE procedure three times to release all the graphics memory in use. The first RELEASE would reclaim the highest 16K bytes, used for buffer 2 of mode 1, 2, or 3. The next RELEASE would reclaim the next 8K bytes, the space occupied by buffer 2 of mode 0. The third RELEASE would reclaim the last 8K bytes, where buffer 1 of mode 0 is stored.

If you were using only 16K bytes of graphics display space, performing RELEASE twice would release all the graphics memory in use. The first RELEASE would reclaim the highest 8K bytes (buffer 2 of mode 0) and the next RELEASE would reclaim the last 8K bytes (buffer 1 of mode 0).

If you were using only 8K bytes (buffer 1 of mode 0), a single RELEASE would reclaim all the graphics memory in use. Extra RELEASE procedures, performed when there is no memory still reserved for graphics, have no effect.

Closing the .GRAFIX Driver

When you issue one of the BASIC statements RUN, LOAD, or NEW, all files currently open are automatically closed. Thus, you do not normally have to explicitly close the .GRAFIX driver.

You may, of course, close the .GRAFIX driver at any time in your program. Closing the .GRAFIX driver does not, by itself, release any of the graphics memory currently in use.

If you re-open the .GRAFIX driver after closing it, all the initial default conditions for graphics are reset. For example, if you used the statement

```
100 OPEN#1, ".GRAFIX"
```

to open the .GRAFIX driver originally, you could later reset all the graphics default conditions with the statements

```
2250 CLOSE#1
2260 OPEN#1, ".GRAFIX"
```

Returning to Text Mode

The BASIC statement

```
)TEXT
```

causes a return from any graphics display to normal, 80-column black and white text mode. The text screen is not cleared, and the text cursor position is not moved by this statement.

If you are using a different text mode such as 40-column color text, you can return from any graphics display to your previously selected text mode, without erasing the screen or moving the text cursor, by using this statement:

```
)PRINT CHR$(15)
```

Unlike TEXT, this statement does not change to the standard, default text mode.

Note that while a graphics buffer is being shown on the screen, any operations that would normally affect the text display still do so. If the program subsequently does a PRINT CHR$(15), the text display will reflect those text operations.

If an error halts your program while you are showing graphics on the screen, you are automatically returned to the previously chosen text mode so that you may see the error message. However, if your program ends while graphics are being displayed on the screen, you must issue the statement TEXT or PRINT CHR$(15) to return to a text-mode display.

Graphics in Display Mode 1

Graphics display mode 1 exists as a by-product of the 40-column color text modes. It can be used for sixteen-color graphics with a horizontal resolution of 280 dots across the screen, but you must be aware of some rather tricky restrictions.

First, imagine that each of the display dots in this mode can only be either "on" or "off". Dots may be turned "on" by plotting dots (DOTAT, DOTREL), lines (LINETO, LINEREL), or the foreground of text characters (PRINT#) and other images (DRAWIMAGE). Dots may be turned "off" by filling operations (FILLPORT), or by plotting the background of text characters or other images.

Now, imagine each 280-dot horizontal screen line as divided into forty segments of seven dots each. Within each seven-dot segment, the "on" dots all appear in one color called the "foreground color", while the "off" dots all appear in another color called the "background color". That's the big restriction.

The color used for the "on" dots in a segment is the pen color used for the last plotting or foreground operation which turned on any dot in that segment. This foreground color may be any of the sixteen colors, but changing the color of any "on" dot within a seven-dot segment simultaneously changes the color of every other "on" dot in that segment.

The color used for the "off" dots in a segment is the fill color used for the last filling or background operation which turned off any dot in that segment. This background color may be any of the sixteen colors, but changing the color of any "off" dot within a seven-dot segment simultaneously changes the color of every other "off" dot in that segment.

Mode 1 can thus be used quite effectively for any graphic display which uses only two colors: a background color and a foreground color. In fact, separate areas of the screen can each contain such a two-color display, using different pairs of colors. Such areas can occupy adjacent dots vertically, but should be separated by at least seven dots horizontally for safety. One horizontal dot separation is sufficient, of course, if you can be sure those dots are in different seven-dot segments. For example, the dot at x=6,y=34 and the dot at x=7, y=5 are in different segments. The dots at x=14,y=2 and x=14,y=3 are also in different segments.

On a single background color, lines may be drawn in any color as long as they remain at least seven horizontal dots from any line in a different color. However, a single line in a third color, when drawn through a fine two-color display, may cause very strange effects over a seven-dot-wide stairstepped area surrounding the line. See the Standard Device Drivers manual for more information and another description of mode 1 graphics.

For your convenience, here are the ranges of x-coordinates which, on a given horizontal line, specify dots in each of the forty, seven-dot segments:

Seg#	Range of x	Seg#	Range of x
1	0-6	11	70-76
2	7-13	12	77-83
3	14-20	13	84-90
4	21-27	14	91-97
5	28-34	15	98-104
6	35-41	16	105-111
7	42-48	17	112-118
8	49-55	18	119-125
9	56-62	19	126-132
10	63-69	20	133-139

Seg#	Range of x	Seg#	Range of x
21	140-146	31	210-216
22	147-153	32	217-223
23	154-160	33	224-230
24	161-167	34	231-237
25	168-174	35	238-244
26	175-181	36	245-251
27	182-188	37	252-258
28	189-195	38	259-265
29	196-202	39	266-272
30	203-209	40	273-279

Creating and Storing a Bit Array

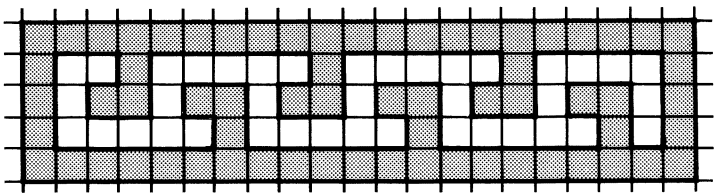
The graphics procedures DRAWIMAGE and NEWFONT both require information which has been stored in the Apple in a certain, known form. This information usually begins as a drawing of an image or character on graph paper, and is finally stored as a series of binary bits (ones and zeros) in memory.

The drawing on graph paper can be viewed as a two-dimensional array of squares, some of them darkened and others left blank. You can then represent this drawing as a two-dimensional array of bits, some of them ones and others left as zeros. Finally, you must convert this array (or "block") of bits to a form which can appear in your BASIC program and be stored in the Apple's memory for the later use of DRAWIMAGE or NEWFONT .

In this section, you will first learn by example how to create a block of bits for use by DRAWIMAGE. After you have read that discussion, the description of the source block for NEWFONT will be much easier to understand. The stored block of bits used by NEWFONT is just a special case of that used by DRAWIMAGE . Internally, in fact, NEWFONT characters are printed on the graphics screen by a specialized use of the DRAWIMAGE routine, using NEWFONT's block of bits.

A Source Block for DRAWIMAGE

Here is a little example that may help you create your own source block of bits for use by the DRAWIMAGE procedure. Suppose you have drawn the following design on graph paper:



Now, to turn your design into a source bit block, put a "1" in each square you want later to appear on the screen in the pen color, and put a "0" in each square you want to appear in the fill color:

Row	Byte 0	Byte 1	Byte 2	Byte 3
0	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 0 0 0	0 0 0 0 0 0 0 0
1	1 0 0 1 0 0 0 0	1 0 0 0 0 1 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0
2	1 0 1 1 0 1 1 0	1 1 0 1 1 0 1 1	0 1 1 0 1 0 0 0	0 0 0 0 0 0 0 0
3	1 0 0 0 0 0 1 0	0 0 0 0 0 1 0 0	0 0 1 0 1 0 0 0	0 0 0 0 0 0 0 0
4	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 0 0 0	0 0 0 0 0 0 0 0

The actual design is only 21 dots wide, but we have extended the bit block out to 32 bits wide. The rows must be extended in one-byte (8-bit) increments, because DRAWIMAGE specifies the row length in bytes. For convenience, however, we are going to assign each row to one or more elements of an integer array, and an integer value is always stored as a two-byte (16-bit) number. For this reason, we extend the rows in two-byte increments. In actual use, we will use DRAWIMAGE to transfer only the first 21 bits of each row, so the extra zeros will not matter.

Next, we must convert each sixteen-bit portion of each bit block row into its decimal equivalent, and assign that decimal number to the appropriate element of our dimensioned integer array (which we will call SOURCE%). Here is a hint about one way to do this conversion on the sample 16-bit number 1001110001111011 :

1. Separate the number into four, 4-bit numbers:
   1001 1100 0111 1011

2. Express each 4-bit number as a hexadecimal digit:
   9 C 7 B

3. Let BASIC convert this 4-digit hexadecimal number into decimal and assign it to an array-variable element:
   470 ARRAY%(4,7) = TEN ("9C7B")

For our example design, these lines would do the job of storing the source bit block in integer array variable SOURCE% :

```
300 DIM SOURCE% (1,4)
310 SOURCE%(0,0) = TEN ("FFFF") : SOURCE%(1,0) = TEN
("F800")
320 SOURCE%(0,1) = TEN ("9041") : SOURCE%(1,1) = TEN
("0800")
330 SOURCE%(0,2) = TEN ("B6DB") : SOURCE%(1,2) = TEN
("6800")
340 SOURCE%(0,3) = TEN ("8208") : SOURCE%(1,3) = TEN
("2800")
350 SOURCE%(0,4) = TEN ("FFFF") : SOURCE%(1,4) = TEN
("F800")
```

The first subscript indicates the horizontal 16-bit integer section within the row, while the second subscript indicates the row number. This order of the subscripts is very important. It is also important that the ROWBYTES given in the DRAWIMAGE procedure is exactly the number of bytes allowed by the maximum first subscript given in the Dimension statement for the two-dimensional integer array variable.

And these lines would serve to place our design on the screen, with the current cursor position at its top left corner:

```
500 ROWBYTES = 4 : XSKIP = 0 : YSKIP = 0
510 DWIDTH = 21 : DHEIGHT = 5
520 PERFORM DRAWIMAGE (@SOURCE%(0,0), %ROWBYTES,
%XSKIP, %YSKIP, %DWIDTH, %DHEIGHT)
```

It was convenient, but not actually necessary, to make each row of our example source block fit neatly into a certain number of sixteen-bit integers. This allowed the array of integers to map fairly easily onto the array of bits.

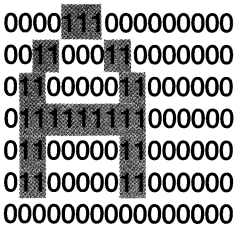
Each source row *must* fit into a certain number of eight-bit bytes, because the ROWBYTES argument of DRAWIMAGE specifies the row length in bytes. If each row is an odd number of bytes in length, you should store the bits in the elements of a one-dimensional integer array. In this case, some of the elements will contain the last eight bits from one source row, and the first eight bits from the next row.

In fact, any source block can be stored in the elements of a one-dimensional integer array. This may be a little more confusing for the programmer, but DRAWIMAGE will not mind: to DRAWIMAGE, every source block appears to be a long, uninterrupted string of zeros and ones. The ROWBYTES argument simply tells DRAWIMAGE how to divide this long string into the rows of your source block.

A Source Block for NEWFONT

The following information may be useful in creating a source block of bits for use by the NEWFONT procedure. This source block is essentially a DRAWIMAGE source block which contains the descriptions of a complete alphabet of characters.

Each character cell in a text font activated by NEWFONT should, at the lowest level, be a source block portion with the following typical arrangement:

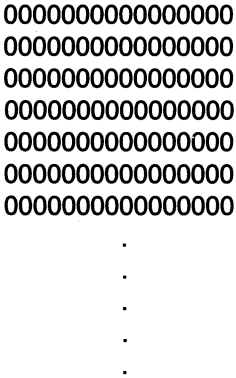


Typical character-cell description.

Each 1-bit in a character cell will later be displayed in the pen color, and each 0-bit will be given the fill color (subject to modification by the color table and the transfer function, of course).

This example shows a character from a font whose character cell is twelve dots wide by seven dots high. Note that each row has been filled out with zeros to make the row two bytes (16 bits) long. Rows must be a whole number of bytes in length, because the new character font is used as a source bit block by the DRAWIMAGE routine, as described earlier in this section.

The set of block portions describing a font of such characters will appear in the complete source block as follows:



Character description corresponding to ASCII code 0.

Character descriptions corresponding to ASCII codes 1 through 64.

```
0000111000000000
0011000110000000
0110000011000000
0111111111000000
0110000011000000
0110000011000000
0000000000000000
0111111110000000
0011000011000000
0011111110000000
0011000011000000
0011000011000000
0111111110000000
0000000000000000
0001111110000000
0011000011000000
0110000000000000
0110000000000000
0011000011000000
0001111110000000
0000000000000000
```

Character description corresponding to ASCII code 65.

Character description corresponding to ASCII code 66.

Character description corresponding to ASCII code 67.

.
.
.

Character descriptions corresponding to ASCII codes 68 through 127.

When PRINT# puts a NEWFONT character on the graphics screen, DRAWIMAGE is automatically performed with a ROWBYTES argument that is the smallest number of bytes needed to contain the number of bits indicated by NEWFONT's character-cell width. This automatic use of DRAWIMAGE also uses an XSKIP of 0, a YSKIP of (ASCII code)*(character-cell height), and a DWIDTH and DHEIGHT which are the NEWFONT character-cell width and height. See the DRAWIMAGE routine for more information.

For character cells 1 to 8 bits wide, each row in the source block must be one byte long (extended with extra bits to fill out the byte, if necessary). Thus, you will have to store two one-byte character rows in each two-byte element of the font's integer array variable. For character cells 9 to 16 bits wide, each source block row must be two bytes long, and can be stored in a single two-byte element of the font's integer array variable. And so on.

Note that rows consisting of odd numbers of bytes do not fit neatly into individual, two-byte integer array variable elements. It is usually easier on the brain to store the source blocks for such fonts in a one-dimensional integer array variable. A source block whose rows are an even number of bytes in width can be stored more easily in a two-dimensional array.

You might also wish to store a font in an array variable with an extra dimension corresponding to the ASCII character code for the character cell descriptions. In that case, the ASCII code would be the last dimension.

NEWFONT can specify character cells with any width and height up to 255 bits by 255 bits, but all of the character cells within one font must be the same width and height. The character rows in the font source block can thus be any width up to sixteen bytes (extending each row in the character cell with extra, unused bits to fill up the last byte, if necessary). A new font may consist of up to 256 character cells, corresponding to ASCII codes 0 through 255.

For easy use, each character description should have the same position in the font source block as it would have in the standard ASCII sequence. This means you should reserve space in the character set array (using all-zero blanks, for instance) even for characters that you will not use (such as the ASCII control characters). You do not need to provide descriptions of characters later in the ASCII sequence than the last character that you will actually use. See appendix A for a description of the standard ASCII character sequence.

The System Font

The standard system character font provides the characters used for all text-mode displays, and is also the default font used for text in graphics displays. Each character cell is 7 bits wide by 8 bits high, so each row in the source block is one byte wide, and storing each character description requires eight bytes.

The bits in the system font source block are arranged differently from the format shown for a NEWFONT source block. Each system font character is described in reverse, left for right, compared to NEWFONT character descriptions. The leftmost bit (MSB) of each byte, rather than the rightmost, is not used. The character description for the letter P, for example, would appear in the system font source block as follows:

```
00011110
01000010
01000010
00011110
00000010
00000010
00000000
```

System font character description corresponding to ASCII code 80.

Direct Control of the Screen

The invokable module BGRAF.INV consists of routines that make using the .GRAFIX driver much easier than it would otherwise be. However, it is also possible to issue commands to the .GRAFIX driver directly. This may be done in addition to using the BGRAF.INV routines, or in place of those routines. Direct screen control commands and BGRAF.INV routines may be intermixed.

Before using the .GRAFIX driver for direct screen control, you must first open the driver and assign it a file reference number, using a statement such as

```
)OPEN#1, ".GRAFIX"
```

This example statement assigned .GRAFIX the file reference number 1, but you could assign it any integer from 1 through 10 which is not already assigned to a file.

Once you have opened the .GRAFIX driver, you can use the driver's file reference number to send a stream of command characters to the driver. Most of these characters are not printing characters, so you will use the CHR$ function to convert each character's ASCII code number into the corresponding character.

For example, if you have already opened .GRAFIX and assigned it the file reference number 1, you could change the pen color to yellow with the following statements:

```
)SETPEN = 19 : YELLOW = 13
)PRINT#1; CHR$(SETPEN); CHR$(YELLOW)
```

Direct screen control is more difficult if the operations require two-byte integer arguments. You must convert such arguments into the form low-byte, then high-byte, and send the character corresponding to the decimal equivalent of each byte. See the *Standard Device Drivers* manual for details about the characters used for direct screen control, and their arguments.

Summary

Starting Up

When you are ready to use graphics, issue BASIC statements such as these:

```
100 OPEN#1, ".GRAFIX"
110 INVOKE "BGRAF.INV"
```

Any file reference number (1 through 10) may be used.

Using Graphics Routines

To use a graphics procedure, issue BASIC statements such as

```
140 PERFORM PENCOLOR (%13)
150 X = 23 : Y = 44
160 PERFORM LINETO (%X, %Y)
```

Every argument which is to be passed to the procedure as an integer must be preceded by a percent sign (%).

To use a graphics function, issue a BASIC statement such as

```
170 COLOR% = EXFN%.XYCOLOR : X% = EXFN%.XLOC
```

Graphics Modes

Mode Number	Mode Description	Buffer Size
0	280 x 192, Black & White	8K
1	280 x 192, 16 Colors (restricted)	16K
2	560 x 192, Black & White	16K
3	140 x 192, 16 Colors (unrestricted)	16K

Display Buffers

Buffer Number	Buffer Name
1	Primary Buffer
2	Secondary Buffer

Colors

Color Number	Color Name	Color Number	Color Name
0	Black	8	Brown
1	Magenta	9	Orange
2	Dark Blue	10	Grey 2
3	Purple	11	Pink
4	Dark Green	12	Green
5	Grey 1	13	Yellow
6	Medium Blue	14	Aqua
7	Light Blue	15	White

Transfer Options

Option Number	Option Name	Result Color
0	Replace	Source Color
1	Overlay	Source Color OR Screen Color
2	Invert	Source Color XOR Screen Color
3	Erase	(NOT Source) AND Screen Color
4	Inverse Replace	(NOT Source) Color
5	Inverse Overlay	(NOT Source) OR Screen Color
6	Inverse Invert	(NOT Source) XOR Screen Color
7	Inverse Erase	Source Color AND Screen Color

The Graphics Procedures

Procedure	Description
INITGRAFIX	Resets four initial conditions: full-screen viewport; cursor at x=0,y=0 (lower left corner); normal color table and transfer function.
GRAFIXMODE (%MODE, %BUFFER)	Selects a graphics mode and display buffer for next graphics operations.
GRAFIXON	Shows graphics on the screen, using mode and buffer currently selected.
VIEWPORT (%LEFT, %RIGHT, %BOTTOM, %TOP)	Sets the viewport boundaries; no operation will plot dots outside the current viewport.

PENCOLOR (%COLOR)	Sets pen color used for plotting dots, lines, and foreground of text characters and DRAWIMAGE blocks.
FILLCOLOR (%COLOR)	Sets fill color used for erasing the viewport and for plotting the background of text characters and DRAWIMAGE blocks.
SETCTAB (%SOURCECOLOR, %SCREENCOLOR, %RESULTCOLOR)	Sets a condition in color table. If any operation tries to plot a dot with pen color or fill color SOURCECOLOR at a point where the existing display dot has the color SCREENCOLOR, dot is given the new color RESULTCOLOR.
XFROPTION (%OPTION)	Sets transfer mode, which may alter color used for plotting any dot. Does bit-wise logical operation on source color for a dot (pen color, fill color, or result of color table) and existing color of that dot in the display to determine a new color for the display dot.
MOVETO (%X, %Y)	Moves the cursor to absolute position x=X,y=Y.
MOVEREL (%DX, %DY)	Moves the cursor DX dots to the right, and DY dots up (negative values move left or down).

DOTAT (%X, %Y)	Plots a dot at absolute position x=X,y=Y in the pen color.
DOTREL (%DX, %DY)	Moves DX dots to the right and DY dots up (negative values move left or down), and plots a dot there in the pen color.
LINETO (%X, %Y)	Draws a line in the pen color from the old cursor position to absolute position x=X,y=Y.
LINEREL (%DX, %DY)	Draws a line in pen color from old cursor position to a point DX dots to right and DY dots up (negative values move left or down).
FILLPORT	Fills the viewport with the current fill color.
NEWFONT (@FONT%(0,0), %WIDTH, %HEIGHT)	Selects new set of characters, stored in integer array variable FONT%, which PRINT# can put into graphics. Each character cell in the new font is WIDTH dots wide by HEIGHT dots high.
SYSFONT	Returns to the normal text-mode font for graphics characters.
DRAWIMAGE (@BLOCK%(0,0), %ROWBYTES, %XSKIP, %YSKIP %DWIDTH, %DHEIGHT)	Puts a predefined image, stored in integer array variable BLOCK%, on the display with cursor at top left corner of the image. Source block is

	ROWBYTES bytes wide. From top left corner of block, skip XSKIP bits right and YSKIP bits down to find top left corner of the desired portion. Transfer portion DWIDTH bits wide by DHEIGHT bits high to the display, using the pen color for 1-bits, fill color for 0-bits.
GSAVE."PATHNAME	Saves the currently selected display buffer on diskette, in a FOTO file named PATHNAME .
GLOAD."PATHNAME	Loads FOTO file named PATHNAME into currently selected buffer, after setting appropriate graphics mode.
RELEASE	Releases the highest graphics memory buffer space for use by BASIC programs. May be repeated to release any remaining buffers.

The Graphics Functions

Function	**Description**
XYCOLOR	Returns the color number for the dot at the cursor position.
XLOC	Returns the x-coordinate of the cursor position.
YLOC	Returns the y-coordinate of the cursor position.

Useful Basic Statements

Statement	Description
TEXT	Switches display on screen from graphics to normal text mode.
PRINT CHR$(15)	Switches display to last text mode selected.
PRINT1; Apple	Prints text characters on the graphics display, starting at the cursor position (if you used OPEN1, .GRAFIX to open the graphics driver). Cursor marks upper left corner of character's rectangular cell.
PRINT1; CHR$(19);CHR$(13)	Sends direct screen-control characters to the .GRAFIX driver.
LOAD, RUN, or NEW	Each of these statements closes any open files, in addition to its other tasks.

Index

Apple III

Apple Business BASIC Reference Manual - Volume 2

apple computer

10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

030-292-A